

Web Development Kit and Client Applications Development Guide



Version 5.3 SP2
January 2006

Copyright © 1994-2006 EMC Corporation. All rights reserved.

Table of Contents

Preface	21
Chapter 1 What is WDK?	23
Terminology.....	23
Conventions.....	23
Documentation resources.....	24
The WDK architectural stack	26
Content Server	29
J2EE 1.3 application server	29
Service layer.....	29
The WDK environment layer.....	29
Presentation model.....	30
Component model.....	30
Application model.....	31
Client.....	32
Approaches to building a WDK client application.....	32
Configuring and customizing a WDK-based application	33
Building a WDK-based application.....	33
Part 1 Configuring WDK Applications	35
Chapter 2 Configuring and Deploying Applications	37
Application structure.....	38
Application layers	38
Application layer contents.....	39
Application layer inheritance	42
Application environments.....	43
Required application directories for custom applications.....	44
Web application archives (WAR files).....	45
How application elements interact.....	46
Configuration overview	47
What is configurable.....	47
Working with XML configuration files	49
General configuration elements	50
Extending XML definitions	51
Scope.....	52
Client environment qualifier	53
Versioning.....	55
Externalizing and configuring strings	56
Configuring an application.....	57
Application name	58
Application configuration file (app.xml).....	58
<config> element	60
<scope> element	60
<application> element	60
<qualifiers> element.....	60

<environment> element	60
<failover> element	62
<fallback_identity> element	62
<language> element	62
<save_credential> element	63
<authentication> element	63
<rolemodel> element	64
<themes> element	64
<accessibility> element	65
<contentxfer> elements	66
<browserrequirements> element	70
<errormessageservice> element	70
<infomessageservice> element	70
<requestvalidation> element	71
<adobe_comment_connector> element	72
<notification> element	72
<session_config> element	73
<xmlfile_extensions> element	73
<formats> element	74
<preferred_renditions> element	75
<modified_vdm_nodes> element	76
<custom_attribute_data_handlers> element	77
<discussion> element	77
<xforms> element	77
<listeners> element	78
<client-sessionstate> element	78
<dragdrop> element	79
<copy_operation> elements	79
<move_operation> elements	80
<richtexteditor> element	80
<plugins> element	80
<display> element	81
<applet-tag> element	81
<job-execution> element	82
Web deployment descriptor (web.xml)	83
Application environment properties	88
Configuring application failover support	89
Configuring application-wide failover	89
Configuring component failover	90
Configuring content transfer mode for the application	91
Virtual links	91
Virtual link handler deployment	92
Virtual link connection and authentication	93
Virtual link path resolution and document delivery	94
Virtual link error handling	96
Content server event notification	97
Navigation defaults	97
Browser history	98
Cookies	99
Timeout	101
Application login and authentication	103
Per-session authentication (login dialog)	104
J2EE principal authentication	104
Single sign-on	107
Ticketed login	108
Skip authentication	110
Explicit login	111
Login preferences	111

Login locale.....	111
Number of user sessions	112
Using events and JavaScript	112
Navigating with an event handler.....	112
Client-side navigation.....	113
Registering client event handlers	114
Using client-side scripts	115
Manual scripts.....	115
Registered scripts	115
WDK scripts.....	116
Generated script tags	116
JavaScript tracing.....	117
Events between frames	117
Inter-frame event handlers	119
Inter-frame server events	119
Managing frames.....	120
Calling JavaScript functions from server-side classes.....	121
Branding an application.....	122
Registering a theme	123
Creating a theme directory	124
Making a theme available.....	125
How themes are located.....	126
Using style sheets	127
Using images in style sheets	128
Default WDK style sheet	129
Internationalized style sheet	130
Modifying a style sheet	130
Identifying styles in WDK applications	130
Adding images and icons	133
Configuring buttons	134
Configuring the file selector applet	135
Branding examples	135
Configuring and localizing strings	137
Adding locales	138
Adding strings to properties files.....	138
Inheriting strings	139
Naming properties files.....	140
Adding localized files to your application	140
Overriding strings in the UI	141
Designing for and testing internationalization	141
Configuring search	144
Configuring search controls	145
Configuring basic search	146
Configuring advanced search	147
Configuring search results.....	151
Making search results configurable by users.....	153
Using 5.2.5 custom search components.....	154
Packaging and deploying Web applications.....	154
WAR packaging tool	155
Deploying with the application installer.....	156
Development update tool.....	156
Compiling and precompiling JSP pages	157
Chapter 3 Configuring Controls	159
What controls do	160
How to configure controls.....	160

Finding files to configure controls.....	162
Using tag libraries	164
Control events.....	165
Types of control events	166
Configuring control events.....	168
Control event arguments.....	168
Handling a control event on the client.....	168
Types of controls	169
Action-enabled controls	171
Types of action controls.....	172
Dynamic action controls.....	172
Using dynamic action controls.....	174
Controlling visibility.....	174
Controls that can be globally configured	175
Hiding controls	177
Configuring dates.....	177
Configuring menus.....	178
Passing arguments to menus or dynamic action controls	180
Configuring tabs.....	181
Configuring dropdown lists	181
Configuring scrollable controls.....	183
Configuring databound controls.....	184
Configuring data display	185
Providing data to databound controls	186
Configuring data sorting	188
Configuring data paging.....	189
JSP fragment control	189
Configuring rich text.....	190
Displaying and validating attributes	192
Single and repeating attributes	193
Displaying lists of attributes	193
The attributelist control.....	194
Context-based attribute lists	195
Attributelist configuration files	196
Using data dictionary attribute lists	199
Supplying or overriding data dictionary attribute lists	200
Display of escaped HTML strings	201
Configuring pseudoattributes.....	201
Validating user input	202
Validator controls	202
Input mask validator	203
Validating an object and its attributes.....	204
Using value assistance	204
Implementing non-data dictionary value assistance	205
Working with images and icons.....	206
Icon controls	207
Using icons	207
Working with tooltips	208
Chapter 4 Configuring Actions	211
What actions do.....	211

	How to launch an action	212
	Adding action controls to a JSP page.....	213
	Passing arguments to actions.....	213
	Generic actions using LaunchComponent	215
	Action configuration file	215
	Precondition permissions.....	218
	Hiding an action for subtypes.....	218
Chapter 5	Configuring Components	219
	Component features	219
	Component configuration file.....	221
	Component inheritance (extends)	224
	Component scope.....	225
	Hiding components	227
	Hiding component features.....	228
	Configuring data columns.....	229
	Adding or removing static data columns.....	229
	Configuring dynamic data columns.....	232
	Component layout (JSP pages)	233
	JSP pages modeled by form class	233
	Contents of a WDK JSP page	234
	JSP includes	235
	Creating a component JSP page	236
	Using messages and labels	236
	Using a raw JSP or static HTML file	237
	Component navigation	237
	Calling components by URL.....	238
	Calling components from an action (LaunchComponent).....	239
	Calling components from JavaScript	239
	Including a component in another component.....	240
	Navigating using browser history.....	241
	Component operations on foreign objects.....	242
	Presubmission client events.....	242
	Configuring containers	243
	Container types	244
	Calling containers.....	247
	Calling a container by URL	247
	Calling a container by JavaScript	247
	Calling a container from an action	248
	Configuring containers	249
	Require visit.....	249
	Container labels.....	249
	Components that must run within a container	250
	Creating modal containers	251
	Configuring locators	251
	Using JSP pages outside a component.....	257
Chapter 6	Configuring Application Connector menus, components, and actions	259
	Overview	259
	Modifying the Documentum menu.....	259

	Overview	260
	Removing menu items from all application connectors.....	261
	Modifying menu items for all applications	261
	Adding custom menu items to all applications	264
	Restricting menu items to specific applications	264
	Customizing application connector components and actions	265
	Overview	266
	List of application connector components and actions	266
	Adding application connector components and actions.....	266
	appintgcontroller component	267
	Managing events	270
	Managing authentication	271
Chapter 7	Configuring Preferences	273
	Preference definition.....	273
	Configuring default component and user preferences	276
	User column display preferences	277
	Sample preference definitions.....	281
Chapter 8	Configuring Roles and Client Capability	287
	Role configuration overview.....	287
	Client capability plugin.....	289
	Docbase role plugin	291
	Role-based actions	292
	Role-based filters	293
	Role-based UI.....	294
Chapter 9	Configuration Examples	295
	Configuring buttons and images.....	295
	Adding a button or image	296
	Changing a button label	296
	Changing a button or image style	297
	Changing button or image function	298
	Configuring dynamic buttons and images	299
	Configuring dynamic menu items.....	300
	Configuring content display	302
	Configuring navigation base cabinet or folder	305
	Configuring branding	307
	Configuring validators.....	308
	Configuring application startup.....	309
	Configuring accessibility	310
	Configuring the properties container	311
	Configuring attributes.....	312
	Configuring attribute layout.....	313
	Configuring an attribute list	314
	Creating a custom object filter	317
Part 2	Customizing WDK Applications	321
Chapter 10	Development Environment and Tools	323

Using an IDE.....	323
Troubleshooting WDK-based applications.....	324
Runtime errors	324
Error loading main component.....	325
Show All Properties does not work.....	325
Properties do not display after data dictionary change	325
WebLogic compiler fails.....	326
WebLogic slows, throws exceptions, or crashes.....	326
(WebLogic) java.io.IOException: Not enough space	327
Future dates do not display correctly	327
JavaScript error on application connection.....	327
Error "Configuration base has not been established"	327
Application no longer starts after code change.....	328
(Tomcat) Application slows down	328
Page not found errors in if HTTP 1.1 not enabled in client browser	328
DFC business object no longer works.....	329
Application runs out of sessions	329
Browser navigation renders actions or links invalid	329
Content transfer fails	330
(Windows) Applet installation fails on client	330
Cannot import an XML file.....	331
Cannot check in XML file	332
java.lang.verify error in WDK application after installing another Documentum product	332
Unable to locate checked out objects after installing WDK-based application	332
(WebLogic) Invalid ticket (content transfer fails)	332
Controls don't display any repository data	333
Tracing	333
Turning on WDK tracing.....	333
Using DFC tracing.....	334
Using DMCL tracing.....	334
WDK tracing flags	335
Tracing sessions.....	336
Tracing WDK framework operations.....	337
Tracing controls and validation.....	337
Tracing JSP processing	338
Tracing components and applications	339
Tracing virtual links.....	339
Tracing servlets	340
Tracing asynchronous operations.....	340
Tracing content transfer	341
Adding custom tracing flags.....	342
Client-side tracing	342
Logging	343
Performance.....	344
Action implementation	345
Documentum object creation	345
String management.....	346
Paging	346
J2EE memory allocation	346
HTTP sessions.....	348
Preferences.....	349
Browser history	349
Value assistance.....	349
Search query performance.....	350

High latency and low bandwidth connections	350
Qualifiers and performance	352
Import performance	352
Load balancing	352
Modal windows	353
Finding component information	353
Comment stripper	353
Testing components	354
Debugging tips	354
Refreshing configuration and data dictionary	355
JSP debugging	355
XML debugging	356
JavaScript debugging	356
Java debugging	357
Chapter 11 Component, Action, and Control Design Guidelines	359
General guidelines	359
File follows naming convention	359
File follows location convention	361
Follows accessibility guidelines (Section 508)	362
Externalizes and tests strings	362
Design checklists	362
Control checklist	363
Control checklist detail	365
Creating new control if needed	365
Using base tag rendering helpers	366
Formating and escaping rendered HTML	366
Ensuring that page is loaded and initialized	367
Component checklist	367
Component checklist detail	373
Describing a component	374
Making a component configurable	374
Making the component definition backward-compatible	374
Removing context-sensitive behavior from the class	375
Caching data	375
Using custom attribute data handlers	375
Following DFC guidelines	376
Component unit test checklist	376
Chapter 12 Customizing Controls	379
Control classes	380
Using controls programmatically	382
Creating controls	383
Naming and getting controls	384
Setting control values	386
Getting datagrid controls	388
Passing arguments to action-enabled controls	389
Programming databound controls	389
Data support classes	390
Getting data	390
Getting data in a component	391
Getting data in a tag class	391
Getting data in a behavior class	392
Getting or overriding data in a JSP page	393
Refreshing data	394

Caching data	394
Modifying the display and handling of attributes	395
docbaseobjectconfiguration file.....	395
Attribute formatters.....	397
Value handlers	398
Tag classes	398
Custom elements and editing components in object configuration.....	401
Default configuration.....	402
DocbaseAttributeList lookup process.....	403
Rendering data with result sets.....	404
Making data scrollable.....	404
Handling data from a configuration file	404
Handling data from an array or vector	405
Formatting data with handlers	406
Adding custom attributes to a datagrid.....	407
Generating UI	409
Generating a link in a control	410
Making a control accessible to JavaScript.....	411
Displaying folder paths and breadcrumbs.....	412
Getting the primary folder path.....	412
Displaying the folder path.....	412
Adding support for a breadcrumb	413
Using a hidden folder path in a component.....	414
Implementing multiple selection	415
Managing control events.....	416
Use server-side or client-side processing?.....	417
Firing a server event from the client.....	417
Handling a control event on the server.....	420
Updating components with client events.....	421
Firing a client event from the server.....	422
Linking controls by events	422
State change events.....	423
How control events are raised	424
Using modal windows.....	425
Setting event handlers programmatically	427
Control lifecycle events.....	428
Validating a control value.....	428
Validating a repository object	429
Adding a control listener.....	430
Creating custom pseudoattributes	431
How controls and tags work together	432
Control arguments.....	433
Chapter 13 Customizing Components	435
Component base class.....	436
Component public interface	436
Navigating within and between components.....	437
Navigating within a component	437
Jumping to a component.....	437
Nesting to another component	438
Returning to the calling component	440
Returning to a component, then jumping to another	440

Navigating within a container	441
Implementing failover support	442
Implementing a component.....	445
Using a component listener	447
Accessing an included component	449
Supporting drag and drop.....	450
Drag and drop support in WDK components	451
Adding drag and drop to a component definition	452
Adding drag and drop to a JSP page	454
Adding drag and drop support to a control.....	455
Troubleshooting drag and drop	456
Customizing containers	457
Calling a container from a server class	457
Implementing container notifications	458
Accessing components within containers	460
Passing arguments in a container.....	462
Multi-repository support.....	463
Replica (mirror), reference, and foreign Objects	464
Adding multi-repository support to a component.....	465
Scoping and preconditioning actions on remote objects.....	466
Session management with multiple repositories	466
Component dispatching.....	467
Component dispatcher servlet.....	467
How components are dispatched.....	467
WDK 5 component bridge.....	468
URL bridge (default)	469
Component lifecycle	469
JSP page processing (form processor).....	470
What the form processor does	470
Form processing sequence.....	471
Processing browser navigation	473
Form navigation operations	473
Form classes.....	475
WebformIncludes class	475
Chapter 14 Using the Configuration Service	479
Configuration service classes.....	479
ConfigService	480
IConfigContext.....	480
Configuration lookup	481
Configuration lookup hooks.....	482
Configuration reader	483
Scope and qualifiers.....	484
Context.....	487
Configuration service process.....	488
Lookup algorithm.....	489
Chapter 15 Customizing actions	491
Preconditions	491
Execution.....	493
LaunchComponent execution classes	496

	Providing action NLS strings.....	497
	Dynamic component launching	498
	Action listeners	500
	Nesting actions.....	503
Chapter 16	Customizing Roles	505
	Role service APIs.....	505
	Custom role plugin.....	506
	Role-based menus	507
Chapter 17	Customizing Content Transfer	509
	Content transfer modes compared	510
	Unified client facilities (UCF).....	513
	UCF on the client.....	513
	Configuring the UCF client	514
	Configuring UCF client path substitution.....	517
	Configuring UCF support for unsigned or non-trusted SSL certificates.....	518
	UCF on the application server	520
	Configuring the UCF application server	521
	Configuring UCF support for chunked transfer encoding.....	522
	UCF logging.....	523
	UCF troubleshooting	524
	UCF process.....	525
	Windows client registry in content transfer	527
	HTTP content transfer.....	529
	Content transfer listeners	531
	Content transfer service classes.....	532
	UCF transfer component customization	533
	Content transfer control initialization.....	534
	Content transfer debugging.....	535
	Using Pre-5.3 content transfer components.....	536
	Streaming content to the browser	537
	Content transfer progress.....	537
Chapter 18	Customizing Authentication	539
	Authentication service	539
	Authentication schemes	540
	Silent login.....	542
Chapter 19	Managing Sessions	545
	Getting a session in a component or action class	545
	Getting a session using SessionManagerHttpBinding	547
	Storing and retrieving objects in the session	549
	Binding and caching in a request thread.....	550
	Application, session, and request listeners.....	550
	IDfSessionManagerEventListener	551
	Session synchronization.....	552

	Session tracing	552
	JSP implicit objects in WDK.....	553
Chapter 20	Customizing Search	557
	Programmatic search value assistance.....	557
	Troubleshooting search	558
	Search class diagrams	560
Chapter 21	Implementing Component and User Preferences	563
	Creating a component preference	563
	Storing and retrieving component preferences.....	566
	Storing and retrieving user preferences	567
	Tracing preferences.....	569
Chapter 22	Other Customizations	571
	Asynchronous action and component execution	571
	Asynchronous action job execution.....	572
	Asynchronous component job execution	574
	Job execution framework	575
	UI in asynchronous processing	577
	Asynchronous process	577
	Branding service.....	578
	Image service APIs	579
	Locale service.....	580
	Retrieving localized strings	581
	Dynamic messages in NLS strings	582
	Adding locale support to custom components	583
	LocaleService APIs	583
	Locale codes.....	584
	Accessibility service.....	584
	Accessibility mode.....	585
	Accessible control labels.....	586
	Event handlers	586
	Image accessibility strings	587
	Accessible tables.....	588
	Applet descriptions	589
	Frame titles	589
	Writing alt tags or label descriptions	590
	Help service	590
	Adding help to a standalone Web application.....	590
	The help component	591
	Adding help for a custom component	591
	Invoking the help	592
	Scoping and filtering the help	593
	Launching help	594
	Localizing help files	595
	Utilities	595
	Clipboard service	595
	Clipboard APIs.....	596
	Using the clipboard in a component.....	597
	Location and refresh	598
	Clipboard Action Filtering	599
	Rendering messages to users	600

	Reporting errors	601
	Version utility	603
	Encoding utilities.....	604
	SafeHTMLString	604
	StringUtil.....	605
	ZipArchive.....	606
	Input mask.....	606
Chapter 23	Using Business Objects in WDK	609
	Calling an SBO method	609
	Using TBOs.....	610
Chapter 24	Customization Examples	613
	Displaying Objects: Datagrid and objectgrid	613
	Creating a component.....	614
	Extending a component	614
	Creating a component definition.....	615
	Adding component parameters to the component class.....	615
	Getting data	616
	Creating the component JSP pages.....	616
	Implementing navigation in a component	616
	Customizing components.....	617
	Displaying a single custom object type (object grid)	617
	Creating the custom grid component definition	618
	Creating the object grid class	619
	Adding custom columns to the display	619
	Adding externalized strings	620
	Launching the object grid component	621
	Getting a component reference in a JSP page	623
	Customizing controls.....	623
	Choosing a control superclass	624
	Adding control events.....	624
	Customizing actions	625
	Adding a custom action	625
	Implementing the action execution class	627
	Action tracing	628
	Custom action execution class with pre- and post-processing	629
	Custom queries and data sources.....	630
	Adding a custom query or data source.....	631
	Populating a dropdown list with a query	632
	Creating a validator	633
	Developing a validator tag	633
	Developing a validator class.....	634
	Using the validator in a component	634
	Creating a qualifier	635
	Using a prompt within a container.....	636

List of Figures

Figure 1-1.	WDK physical layout.....	27
Figure 1-2.	Web application architectural stack.....	28
Figure 2-1.	Application layers and configuration inheritance.....	39
Figure 2-2.	Folder tree frame interaction	118
Figure 2-3.	Custom Theme.....	125
Figure 2-4.	Webtop classic view styles.....	131
Figure 2-5.	Webtop streamline view styles	132
Figure 2-6.	NLS strings test	142
Figure 2-7.	Far Eastern characters test.....	143
Figure 2-8.	Long strings test.....	143
Figure 2-9.	Search size custom dropdown list.....	146
Figure 2-10.	Attribute selection dropdown.....	150
Figure 2-11.	Specific attributes as search criteria.....	150
Figure 2-12.	Custom attributes as search criteria.....	151
Figure 2-13.	Custom search results for custom type.....	153
Figure 3-1.	Scrollable pane controls	184
Figure 3-2.	Number of columns in a datagrid	185
Figure 3-3.	DocbaseAttributeList population.....	195
Figure 3-4.	How the configuration service determines attribute list source	200
Figure 5-1.	Scoped configuration.....	226
Figure 5-2.	Root (cabinet) locator.....	253
Figure 5-3.	Flatlist locator	254
Figure 5-4.	Container locator.....	254
Figure 7-1.	WDK preferences components	279
Figure 7-2.	Column selector component.....	280
Figure 7-3.	Custom type column preferences.....	281
Figure 9-1.	Custom type icon	296
Figure 9-2.	Webtop view menu.....	300
Figure 9-3.	Revised view menu	301
Figure 9-4.	Drilldown More... menu	301
Figure 9-5.	Reconfigured More... menu.....	302
Figure 9-6.	Webtop My Files default display.....	303
Figure 9-7.	Configured My Files display	304
Figure 9-8.	Custom attributes display based on context.....	305
Figure 9-9.	Default navigation from repository root.....	306
Figure 9-10.	Navigation from a specific folder path	306
Figure 9-11.	Navigation from a specific folder ID	307

Figure 9–12.	Custom theme directory.....	307
Figure 9–13.	New default theme	308
Figure 9–14.	Adding a component to the properties container	312
Figure 9–15.	Default checkin attributes for a custom type	315
Figure 9–16.	Custom checkin attributes for a custom type	317
Figure 9–17.	Object list with custom filter.....	319
Figure 9–18.	Object list with standard files filter.....	319
Figure 12–1.	String attribute rendered as text control	399
Figure 12–2.	String attribute rendered as TextArea control	401
Figure 12–3.	Docbaseattributelist lookup.....	403
Figure 12–4.	Folder path display.....	413
Figure 12–5.	Client-side and server-side event processing	424
Figure 12–6.	Control and ControlTag relationship.....	433
Figure 13–1.	Serialization process	442
Figure 13–2.	Component Processing Sequence Diagram	471
Figure 13–3.	JSP page, control, and user interaction diagram	472
Figure 17–1.	UCF sample client configuration mapping	514
Figure 17–2.	UCF client-server process.....	526
Figure 17–3.	UCF client-server session management	527
Figure 17–4.	Content transfer component classes and service layer	533
Figure 18–1.	Authentication service interfaces	540
Figure 18–2.	Authentication scheme processing.....	541
Figure 20–1.	Search component UML diagram	561
Figure 21–1.	Component without preference	565
Figure 21–2.	Component with preference	565
Figure 22–1.	Job execution interaction diagram.....	578
Figure 23–1.	Documentum object hierarchy.....	611
Figure 24–1.	Custom type object grid	617
Figure 24–2.	Custom grid launched by URL	622
Figure 24–3.	Launching from the Webtop browser tree.....	622

List of Tables

Table 2-1.	Directories required for Web applications.....	45
Table 2-2.	Binding scenarios for versioned components	55
Table 2-3.	Client environment elements (<clientenv> and <clientenv_structure>).....	61
Table 2-4.	Server environment elements (<serverenv>)	61
Table 2-5.	Language elements (<language>).....	62
Table 2-6.	Save credentials elements (<save_credentials>)	63
Table 2-7.	Authentication elements (<authentication>).....	64
Table 2-8.	Theme elements (<themes>)	65
Table 2-9.	Accessibility elements (<accessibility>).....	65
Table 2-10.	Content transfer common elements	66
Table 2-11.	Content transfer ACS elements	66
Table 2-12.	Client applet content transfer elements (<contentxfer>. <client>).....	67
Table 2-13.	Server content transfer elements (<contentxfer>.<server>).....	69
Table 2-14.	Browser requirement elements (<browserrequirements>).....	70
Table 2-15.	URL request validation elements (<requestvalidation>).....	71
Table 2-16.	PDF Annotation Services elements (<adobe_comment_connector>).....	72
Table 2-17.	Event notification elements (<notification>)	73
Table 2-18.	Session management elements (<session_config>).....	73
Table 2-19.	XML extensions elements (<xmlfile_extensions>).....	74
Table 2-20.	Formats elements (<formats>).....	74
Table 2-21.	Preferred renditions elements (<applications> and <renditions>).....	75
Table 2-22.	Modified VDM action timeout (<modified_vdm_nodes>).....	77
Table 2-23.	Collaborative Edition elements (<discussion>)	77
Table 2-24.	Business Process Manager Forms Builder elements (<xforms>)	78
Table 2-25.	Listener elements (<listeners>).....	78
Table 2-26.	Client session state elements (<client-sessionstate>).....	78
Table 2-27.	Drag and drop elements (<dragdrop>)	79
Table 2-28.	Copy operation elements	79
Table 2-29.	Move operation elements	80
Table 2-30.	Rich text editor elements (<richtexteditor>).....	80
Table 2-31.	Active-X plugins elements (<plugins>)	80
Table 2-32.	Applet tag elements (<applet-tag>)	82
Table 2-33.	Asynchronous job elements (<job-execution>).....	83
Table 2-34.	Context parameters	84
Table 2-35.	WDK filters	85
Table 2-36.	Deployment descriptor listener.....	86
Table 2-37.	WDK servlets	86

Table 2–38.	<errorpage>.....	87
Table 2–39.	Environment settings.....	88
Table 2–40.	Navigation settings.....	97
Table 2–41.	Preferences cookies.....	99
Table 2–42.	Internal cookies.....	100
Table 2–43.	webforms.css (WDK layer).....	129
Table 2–44.	Styles in Webtop classic view and menus.....	131
Table 2–45.	Styles in Webtop streamline view.....	133
Table 3–1.	WDK tag libraries.....	164
Table 3–2.	Event attributes.....	167
Table 3–3.	State of a control cased on dynamic attribute value.....	173
Table 3–4.	Control visibility based on context.....	175
Table 3–5.	Control global configuration.....	175
Table 3–6.	Global date control configuration elements.....	177
Table 3–7.	Rich text configuration elements.....	191
Table 3–8.	Rich text editor configuration elements (<editor>).....	192
Table 3–9.	Sample Documentum Application Builder scope definitions.....	196
Table 5–1.	Types of WDK containers.....	244
Table 6–1.	Application Connectors menu configuration elements.....	262
Table 6–2.	Appintgcontroller <dispatchitems> elements.....	268
Table 6–3.	Required pages in appintgcontroller component definition.....	269
Table 7–1.	<preference> elements.....	274
Table 7–2.	User preference components.....	276
Table 7–3.	Column display preference elements.....	278
Table 8–1.	Client capability roles.....	289
Table 11–1.	Control checklist.....	364
Table 11–2.	Component design checklist.....	367
Table 11–3.	JSP page checklist.....	369
Table 11–4.	Behavior class checklist.....	370
Table 11–5.	Internationalization checklist.....	373
Table 11–6.	Unit test checklist.....	376
Table 11–7.	Component functional testing checklist.....	378
Table 12–1.	Control class properties.....	380
Table 12–2.	Base tag classes.....	381
Table 12–3.	Default attribute handling.....	402
Table 12–4.	Folder path display rules.....	413
Table 13–1.	Components that support drag and drop.....	451
Table 13–2.	Configuration elements (<dragdrop>).....	452
Table 13–3.	WDK target actions.....	455
Table 13–4.	Form navigation operations.....	474
Table 13–5.	URL parameters.....	476
Table 17–1.	Feature support in content transfer modes.....	510
Table 17–2.	How client configuration settings map in content transfer modes.....	511

Table 17-3.	How server configuration settings map in content transfer modes	512
Table 17-4.	UCF client configuration settings.....	515
Table 17-5.	UCF application server configuration settings	522
Table 17-6.	Content transfer registry keys used by content transfer applets.....	528
Table 17-7.	UCF content transfer result variables	531
Table 22-1.	Device-independent events	586
Table 24-1.	Choosing a control superclass	624

Preface

This guide is intended for two tasks:

- **Configuration**

Changes to XML files or modifications of JSP pages to configure controls on the page. Does not require a developer license.

- **Customization**

Extending WDK classes or modifying the JSP pages to include new functionality. Requires a developer license.

Part 1 of this guide describes general configuration of WDK applications. *Web Development Kit Reference Guide* provides additional configuration information on specific controls, actions, and components.

To configure WDK-based applications, you must be familiar with the following technologies:

- JavaServer Pages technology, including tag libraries, in the version supported by your application server
- Cascading style sheets (CSS)
- HTML, particularly forms, tables, and framesets
- JavaScript, including client events and event handling, frame referencing, and form action methods
- XML

Part 2 of this guide describes customization of WDK applications. *Web Development Kit Reference Guide* provides additional configuration information on specific controls, actions, and components.

To customize WDK-based applications, you must be familiar with the above-mentioned technologies and the following additional languages and standards. Customization requires a developer license.

- Java 1.4.x
- J2EE Java Servlet technology, in the version supported by your application server
- Portlet Specification (JSR 168) (WDK for Portlets only)

Revision history

The following changes have been made to this document:

Revision history

Revision Date	Description
March 2005	Initial document release for WDK version 5.3
August 2005	Added information on new features: new configuration for search, date controls, panesets, app.xml, DRL anonymous support, ACS support
September 2005	Added definitions of configuration and customization
January 2006	Added information on search configuration and customization; changes to app.xml; UCF configuration changes; fixed documentation errors

What is WDK?

The Documentum Web Development Kit (WDK) is a Web application tool set. WDK provides the following functionality:

- A Java tag library of easily configured Web-based UI widgets
- A Java framework that supports application-server based state management, messaging, branding, history, internationalization, and content transfer
- A set of configurable components that generate HTML widgets and provide access to repository functionality

WDK's architecture incorporates two models: A presentation model that uses JSP tag libraries to separate Web page design from behavior, and a component model that encapsulates repository functionality in configurable server-side components.

This introduction to WDK includes the following topics:

- [Terminology, page 23](#)
- [Documentation resources, page 24](#)
- [WDK foundation technologies, page 25](#)
- [The WDK architectural stack, page 26](#)
- [Approaches to building a WDK client application, page 32](#)

Terminology

APP_HOME is the root directory of the Documentum Web application in your installation. The paths for control, action, and component files are shown relative to this base location.

Conventions

This guide uses the following conventions:

Convention	Description
<i>Italics</i>	Represents a variable name for which you must provide a value, or a defined term
<code>typewriter</code>	Represents code samples, commands, user input, and computer output
{curly braces}	Indicates a Java or CSS code implementation
<XXX>	Represents an XML element or JSP tag as it appears in an XML or JSP file. End tags are not always included in examples, unless the element is closed, for example, <dmf:webform/>.
notdefined="true"	All values for control and action attributes or component parameters are passed as strings, even though some are treated as boolean values by the control, action, or component class. For example, true and false are treated as Booleans.
<nlsid>MSG_XXX</nlsid>	If you provide National Language Support (NLS), enclose the value keys in <nlsid> and </nlsid> tags. The WDK locale service will replace the value with the corresponding lookup value in the appropriate localized resource file. The user's locale will determine which localized version of the string is used.

Documentation resources

The Documentum Web Development Kit contains documentation and source files to assist you in developing custom Web applications.

- *Web Development Kit and Client Applications Development Guide*
(The current guide) Contains general configuration, customization, and application-building information for application developers.
- *Web Development Kit Reference Guide*
Contains information about all of the configurable settings for controls, actions, and components in WDK, with some additional component-specific customization notes.
- *WDK for Portlets Development Guide*
This guide contains information about configuring and customizing WDK for Portlets
- *Web Development Kit and Applications Tutorial*

This tutorial contains several modules on setting up a development environment and configuring and customizing WDK

- *Web Development Kit and Applications Installation Guide*

This guide describes how to prepare for, install, and deploy WDK and custom WDK-based applications

- *Web Development Kit Release Notes*

This publication contains information on the supported environments as well as known bugs, limitations, and technical notes

- Source files for basic WDK controls and samples are installed in `/wdk/src`.

Source files for all webcomponent layer components and actions are installed in `/webcomponent/src`. For Webtop installations, source files for Webtop components and actions are installed in `/webtop/src`.

- Javadoc API references

This documentation is installed by the WDK installer and includes the Javadoc API reference sets for the WDK, webcomponent, and Webtop Java libraries. The DFC installer, which runs at the beginning of the WDK installer, offers the option of installing the DFC Javadocs.

- The Documentum CustomerNet Web site, support.documentum.com

This Web site provides WDK and client applications support forums, developer tips and component library, sample code, white papers, and a wealth of information to assist you in developing Documentum-enabled applications. Source code for WDK tutorials and some examples from the current guide are provided on this Web site.

Check the Documentum technical support Web site (support.documentum.com) for revisions of the documentation. Click the Documentation link to search for documents related to your installed version of WDK or WDK client application. The support site also provides peer support forums that are monitored by technical support experts.

WDK foundation technologies

The WDK programming model is based the following technologies:

- XML configuration

Components and actions in WDK are configured through XML configuration files. The WDK configuration service reads configuration elements, both WDK-supplied and user-defined. Configuration files make it easy to change the behavior of components, actions, and applications through simple text editing.

- JavaServer Pages Technology

A JSP page is a text file that describes how to process an HTTP request to create an HTML response. A JSP page in WDK consists of fixed (template) HTML and dynamic content rendered by JSP tags, expressions, and scripting. Most of the UI is generated by JSP tags that can be configured on the JSP page. JSP pages are compiled into servlets (Java classes) by the JSP container or by a third-party compiler. These servlets execute on the server when a JSP page is requested. The servlet performs a server task or generates dynamic content that is then displayed on the client browser.

- J2EE Servlet Technology

A Web application runs in a J2EE-compliant JSP container, which provides the Java Runtime Environment (JRE) and, usually, the JSP translator (compiler). Each JSP page is translated into a servlet class and instantiated every time the JSP page is requested. Additional WDK servlets provide back-end support for timeout, content transfer, and virtual link redirection.

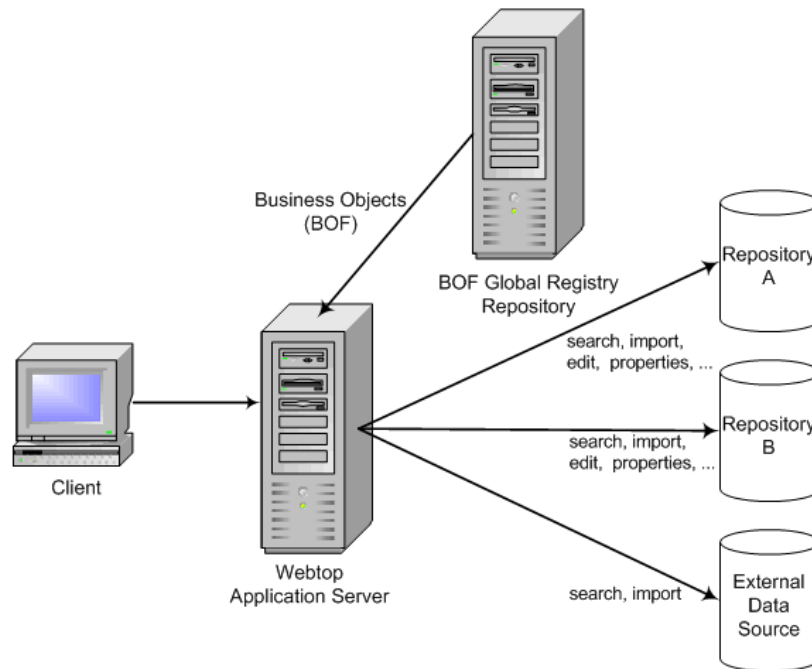
- J2EE security

If you set up J2EE security in your J2EE server, you can configure WDK to support single login. The Java authentication mechanism is used to support sign-on to both the Web server and the repository. Manual authentication, which has been used for previous versions of Documentum clients, is also supported.

The WDK architectural stack

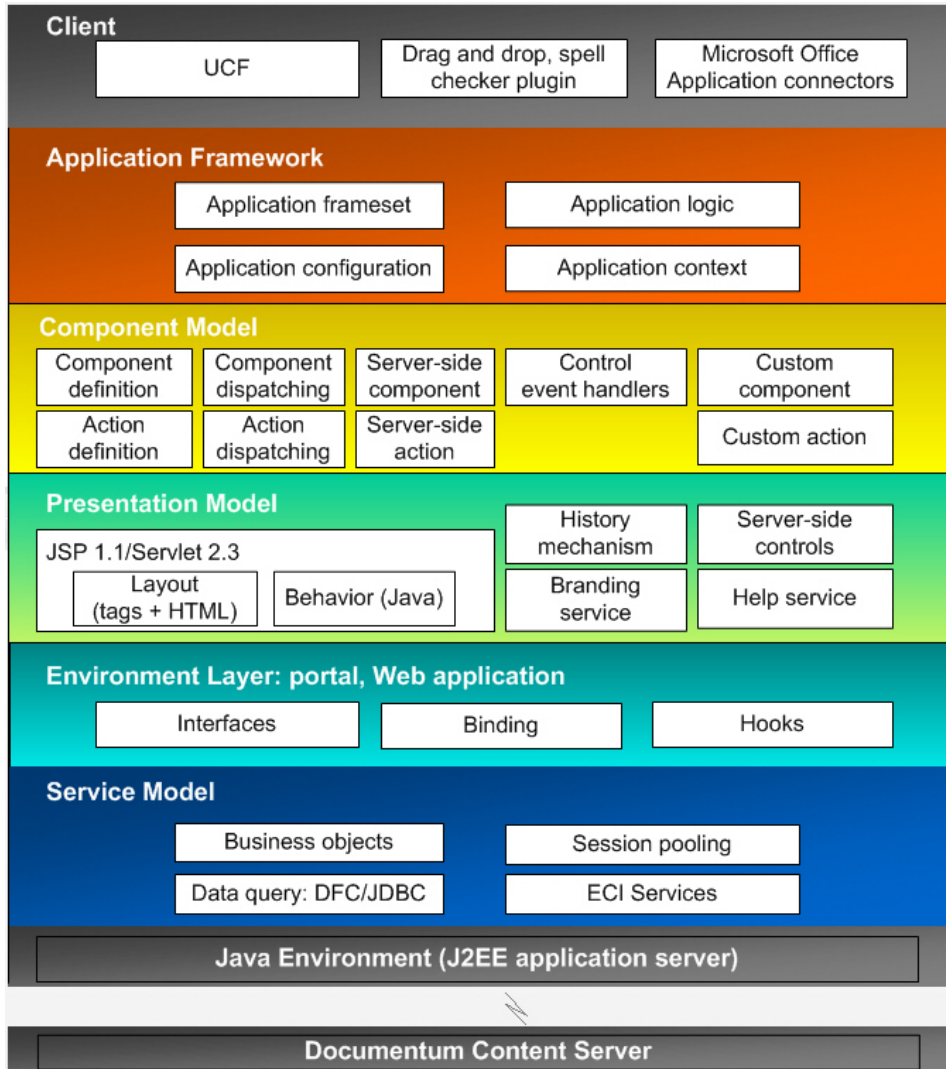
The physical components of a Documentum stack are shown below:

Figure 1-1. WDK physical layout



The components on each host in the stack is illustrated below. The layers in the stack are described in order, starting from the bottom layer.

Figure 1-2. Web application architectural stack



The application stack is described in the following topics:

- [Content Server, page 29](#)
- [J2EE 1.3 application server, page 29](#)
- [Service layer, page 29](#)
- [The WDK environment layer, page 29](#)
- [Presentation model, page 30](#)
- [Component model, page 30](#)
- [Application model, page 31](#)
- [Client, page 32](#)

Content Server

The Web architecture stack has at its base the Documentum Content Server. All of the WDK services and programming model exist to expose the content management functionality of the Content Server.

J2EE 1.3 application server

The J2EE application server provides J2EE classes that are required for HTTP and servlet operation as well as JSP processing and dispatching. The application server requires a Java SDK, and the version of the SDK that is certified by Documentum for each application server is listed in the release notes.

Service layer

The Java environment, DFC, and JDBC connectors provide the following services:

- Data access

The DFC session interface and JDBC connectors provide data access.

- Business Object Framework (BOF)

The DFC BOF places business logic in reusable components.

- Session pooling

The dmdl.ini file on the WDK host can be configured as a client for server connection pooling. WDK applications take advantage of session pooling, increasing performance over non-pooled sessions.

The WDK environment layer

The environment layer in the WDK framework provides a means of supporting various types of Web application environments:

- Standalone Web application, such as Webtop, Documentum Administrator, Digital Asset Manager, or Web Publisher
- JSR-168 compliant portal environments (refer to release notes for certified versions)
- Non JSR 168 compliant portal environments

Presentation model

The presentation model consists of JSP tag libraries and HTML in JSP pages as well as server-side presentation framework.

The Documentum JSP tags generate HTML widgets and databound tables, lists, and other presentation scripting to the browser. Server-side control classes provide access to the control tags and maintain state on the server. A form processor maintains the HTML form state and lifecycle, which is not possible with standard HTML forms.

In addition to the control tags and server-side control classes, the presentation model incorporates the following services:

- Form processor, which interprets HTTP requests and translates requests into WDK method calls and events
- History mechanism, which maintains browser history and navigation
- Configuration service, which looks up configuration contracts for actions and components and dispatches the appropriate UI for the user's context
- Branding service, which manages the look and feel for the application
- Locale service, which delivers a localized UI
- Help service, which delivers localized, context-sensitive help
- Message service, which displays localized messages to the user
- Configurable drag and drop support

Component model

The component model provides a configurable, encapsulated set of Documentum functions or *components*. A component is composed of one or more JSP pages, supporting behavior classes, and an XML configuration file. Component JSP pages use WDK controls and actions from the tag libraries, and each component handles control events with its own event handlers.

The WDK framework enforces a contract for each component, consisting of parameters that initialize the component. The component behavior class includes event handlers that respond to user action and properties that get and set the state of a component.

The component contract is defined in an XML component configuration file. The component is defined within `<component></component>` elements in the file. In addition to contract parameters, the definition includes a component behavior class, an NLS properties resource bundle, a help context ID, and, sometimes, additional configuration elements. A component can include other components, acting as a container.

The component dispatcher dispatches a particular component dynamically on the following criteria:

- Calling context
Context consists of runtime conditions such as object type, current component, or user role. For example, one component definition is called when the selected object is a folder, and another component definition is called when the selected object is a document.
- Component implementation
Several types of component implementation are supported, such as raw JSP pages, WDK 5, or other as specified in the component XML file. For example, if the component is a raw JSP page, the component is dispatched using the J2EE server framework without calling the WDK 5 framework.

Components are often launched by actions. Actions launch components through UI widgets such as menu items or links. An action can evaluate preconditions to ensure that the action is valid for the user's context.

Application model

A Documentum Web application consists of a set of components and the WDK application framework, DFC, and native libraries. The WDK application framework consists of services that apply across the application, such as the configuration, action, messaging, branding, and tracing services.

Documentum's WDK based components are designed to be used in the following application environments:

- Application server
An environment for running JSPs, Servlets and EJBs. Documentum's Webtop application is built for this environment. Services such as authentication and configuration are provided by the WDK framework.
- Portal server
An environment for running portlets, JSP pages, and servlets. Services such as authentication and configuration are provided by the portal server. For more information on portal applications, refer to *WDK for Portlets Development Guide*.
- Application Connectors
WDK provides connectors that enabled Documentum functionality within specific Windows applications such as Microsoft Word. For information on customization Application Connectors, refer to *Application Connectors Software Development Kit Guide*.

The J2EE-compliant root context (base Web application directory) can contain application layers that inherit application parameters from other application layers. For example, the base application layer is WDK. The WDK application layer is extended by the

webcomponent application layer, which in turn is extended by the Webtop application layer. Your custom application layer can then extend the Webtop application layer. The application model enforces consistent appearance and behavior across all application layers contained within the root WDK-based application.

Using branding, an application layer can supply themes that provide your application's unique appearance through icons, images, and style sheets.

Client

Several optional components can be installed on the client:

- UCF
A small-footprint UCF applet is downloaded to the client, and it initiates the download of further client-side support for content transfer
- An Internet Explorer plugin for Windows desktop drag and drop and rich text spell checking can be installed on the client. The availability of this plugin can be configured in in the application configuration file
- Application Connectors to content authoring applications such as Microsoft Word or Excel can be installed on the client, supporting access to repositories and Webtop functionality from within the application

Approaches to building a WDK client application

Documentum Web client applications are built on WDK. Webtop is a reference implementation of WDK. Documentum Administrator, Digital Asset Manager, Web Publisher, and other client applications are extensions of Webtop. WDK for Portlets is an application built on WDK.

Your custom Web application can extend one of the WDK clients such as Webtop or Web Publisher or it can be a new application based on WDK. To design a Web application that incorporates Documentum functionality, use one of the following approaches:

- [Configuring and customizing a WDK-based application, page 33](#)
- [Building a WDK-based application, page 33](#)

Configuring and customizing a WDK-based application

You can configure a client application such as Webtop or Web Publisher by extending client components. You can also add your custom components to the existing client application. You can apply your own corporate branding to your extended application and override default UI and error message strings.

Configuration of a WDK-based applications includes one or more of the following:

- Making textual changes to the application, such as changing its branding, locale, strings, and app.xml settings
- Making changes to any of the application's XML or JSP files that do not require Java class changes
- Modifying existing component definitions without changing the behavior class

Note: You should make these changes to extended component files in the custom folder, not to the original files as installed by the installer.

Customization includes one or more of the following:

- Making changes that extend or implement the application's Java code including component and control classes
- Adding new components or controls to the application
- Making changes that add custom Java classes to the application
- Building new Web applications

To configure or customize an existing WDK-based application, run the WDK installer and select the option **Customize an existing application**. The installer then copies your application to a new directory in the J2EE server directory structure and adds commented content files to the application and adds developer documentation to your system drive. The installer also copies native libraries to your J2EE server host, outside of the J2EE server application directories.

Note: When you customize an existing WDK client application, that application must have the same version of WDK as the current installer.

Building a WDK-based application

To build a new WDK-based application, you must first run the WDK installer and select the option to **Create a new Web application**. The required files are installed in your J2EE server. You must then design and implement an application frameset that ties

together the components in WDK that you wish to use. Refer to *Web Development Kit and Applications Tutorial* for a simple application based on WDK.

Configuring WDK Applications

This section of the development guide describes the configuration methodology for WDK-based Web applications. The following chapters describe WDK applications and configuration:

- [Chapter 2, Configuring and Deploying Applications](#)
- [Chapter 3, Configuring Controls](#)
- [Chapter 4, Configuring Actions](#)
- [Chapter 5, Configuring Components](#)
- [Chapter 8, Configuring Roles and Client Capability](#)
- [Chapter 9, Configuration Examples](#)

Configuring and Deploying Applications

WDK-based Web applications are J2EE-compliant and contain certain WDK framework and component directories. WDK Java and type libraries are contained within the application /WEB-INF directory. The application also requires Documentum native libraries and DFC Java libraries installed in the Documentum home directory on the J2EE server.

This section contains the following information about how to configure and deploy Documentum Web applications:

- [Application structure, page 38](#)
- [Configuration overview, page 47](#)
- [Configuring an application, page 57](#)
- [Application login and authentication, page 103](#)
- [Using events and JavaScript, page 112](#)
- [Branding an application, page 122](#)
- [Configuring and localizing strings, page 137](#)
- [Configuring search, page 144](#)
- [Packaging and deploying Web applications, page 154](#)

For information on migrating WDK 5 applications to newer versions, refer to the [changed APIs document](#) on the Support Web site.

For information on how to make an application accessible to visually impaired users (section 508-compliant), refer to [Accessibility service, page 584](#).

For information on how to set up a development environment for customization, refer to [Chapter 10, Development Environment and Tools](#). A sample development environment is described in *Web Development Kit and Applications Tutorial*.

Application structure

The following topics describe the structure of a WDK-based Web application:

- [Application layers, page 38](#)
- [Application layer contents, page 39](#)
- [Application layer inheritance, page 42](#)
- [Application environments, page 43](#)
- [Web application archives \(WAR files\), page 45](#)
- [How application elements interact, page 46](#)

Application layers

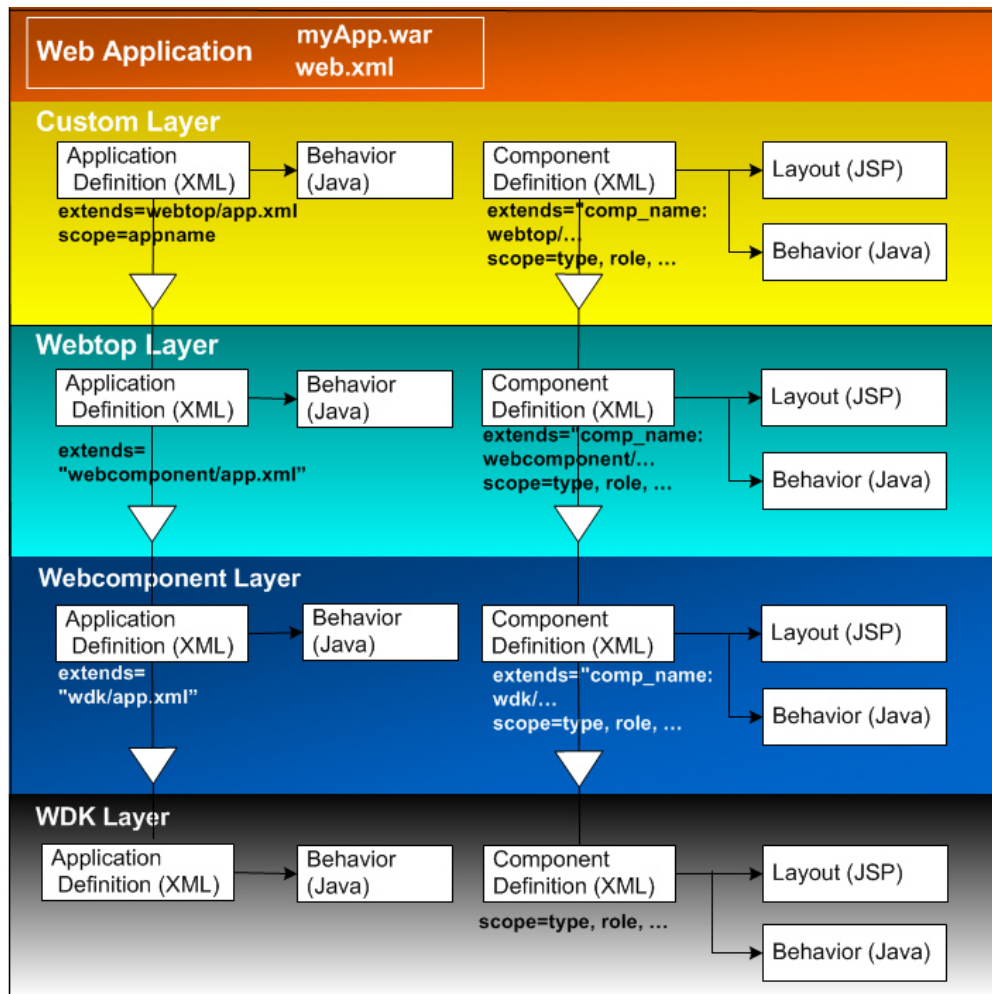
A WDK-based application has a root directory that contains a WEB-INF directory and application layer directories such as /wdk, and /webcomponent. Documentum WDK client applications add the application layer directories /webtop, /wp, /dam, and /da (in various combinations, depending on the client application).

The application directory /custom is provided for your customizations. Place your configuration files, custom JSP pages, NLS resource files, and custom themes in the /custom directory or in subdirectories of /custom. The custom directory serves as the top layer for the application.

Application layers are managed by the WDK framework. Your custom application layer should extend the application definition of the top-level application layer and then add or override functionality. For example, in Webtop, the top application layer that contains application resources is /webtop. If you wish to configure or customize Webtop, use the /custom application layer, which extends the webtop layer. (The app.xml application definition in /custom extends the app.xml application definition in /webtop.) You can simply use the app.xml file in the /custom directory. For more information on application and configuration inheritance, refer to [Application layer inheritance, page 42](#).

A diagram of application layers and their inheritance is shown below.

Figure 2-1. Application layers and configuration inheritance



Application layer contents

The top level directory for each application layer contains the following directories and files. The directories listed are the common or required directories in an application layer. The paths shown are relative to the Web application root directory. Unless otherwise noted, these files and directories are required by a WDK-based application.

- `app.xml`

Application configuration file. Can extend another application's `app.xml` file. Contains application-wide behavior classes such as qualifiers, servlets, and listeners. The application name is applied as the scope

- /config
Configuration files for the application layer components
- /include
(optional) JavaScript files that are used within the application layer or within applications that extend the application
- /strings
Externalized strings
- /theme: Directory that contains themes for the application

The following list describes the directories and files that are specific to each application layer:

- Root directory: Top-level directory for the Web application.
 - default.html and index.html (identical): Entry point to a WDK or Webtop application
 - drl.html: Used by the DRL component to display content referenced by a DRL
 - unstripped.jar: Contains the entire WDK application with all comments intact. In the deployed WDK application, the comments and spaces are stripped out.
 - version.properties: Contains the WDK version number
- /custom: Directory for your customizations. This directory should contain your action and component configuration files (/config subdirectory), JSP pages, branding directories (/theme subdirectory), and externalized strings (/strings subdirectory). Custom classes should be placed in a subdirectory of /WEB-INF/classes using the Java package and directory convention.
- /help: Contains help files for WDK-based applications
- /META-INF: Contains the J2EE manifest file
- /WEB-INF:
 - web.xml: File required by J2EE Web applications. Specifies application startup directory and application-specific servlets.
 - web.xml.authenticate: Same as web.xml, with additional J2EE security element
 - /classes: Contains Java classes that are used by the Web application, including classes for application layers wdk, webcomponent, and webtop (if installed), and your custom application. Also contains some property files (except for NLS property files, which are located in each application layer /strings directory).
 - /lib: Contains Java application archive (JAR) files required by WDK-based applications
 - /tlds: Contains Documentum and custom JSP tag libraries

- /wdk: Contains the following directories and files:
 - Base JSP pages for login, navigation, modal dialogs, timeout, and tracing
 - /container: JSP pages for container components
 - /contentXfer: WDK 5.2.5 content transfer applets, for backward compatibility
 - /control: Reusable JSP pages that serve as mega-controls to display attributes
 - /css: Contains a style sheet for the drag and drop plugin
 - /fileselector: File selector applet used in both WDK 5.2.5 and UCF content transfer
 - /fragments: Contains environment-specific JSP fragments that allow your JSP pages to run in multiple environments (Webtop, portals, or Application Connectors)
 - /images: Images and icons used in the UI
 - /include: JavaScript used by WDK-based applications
 - /logos: Contains the Documentum logo
 - /native: Contains the plugin files
 - /samples: Sample JSP pages that exercise the controls in WDK
 - /src: Source files for the basic controls and sample pages (installed by the WDK installer)
 - /strings: NLS resource files containing all strings in this layer
 - /system: Component JSP pages
 - /theme: CSS files and images used in the application
 - /webwfm: Applets and icons for the Web workflow manager
- /webcomponent
 - app.xml: webcomponent layer configuration file
 - /admin: Contains administration component JSP pages
 - /componentList: Displays information about all components in the application
 - /environment: Clipboard and preferences services JSP pages
 - /finder: Contains the finder component
 - /install: DQL scripts to install subscription and administration object types in your repositories
 - /library: Component JSP pages
 - /navigation: Navigation component JSP pages
 - /src: Source files for all components in this layer (installed by the WDK installer)

- /strings: NLS resource files containing all strings in this layer
- /testbed: JSP files for the testbed component
- /theme: CSS files and images used in the application
- /xforms: Contains JSP pages for the Business Process Manager add-on Forms Builder

Application layer inheritance

Each application layer within the Web application must have an XML configuration file named `app.xml` at the root level of the application layer. For example, the Webtop layer has an `app.xml` file in the `/webtop` directory. This configuration definition extends the definition found in the webcomponent layer application configuration file (`/webcomponent/app.xml`), which extends the definition in the wdk layer application configuration file (`/wdk/app.xml`). Each application layer can add its own parameters and override some of the inherited parameters from parent application layers.

The configurable elements in `app.xml` are described in [Application configuration file \(app.xml\)](#), page 58.

You can have multiple application layers. Each application must have a root directory at the level of `/WEB-INF`. Each application layer must extend another application layer definition and must have an `app.xml` file and a `/config` directory. You cannot have two layers that extend the same layer. For example, you could have a company application layer and a marketing division application layer, with the following structure under the root Web application:

```
/ my_app (root)
  /help
  / marketing
    app.xml application extends="my_company/app.xml"
    /config
  / my_company
    app.xml application extends="webcomponent/app.xml"
    /config
  /webcomponent
    app.xml application extends="wdk/app.xml"
    /config
  /wdk
    app.xml
    /config
  /WEB-INF
```



Caution: Do not add your custom application files to the `/wdk`, `/webcomponent`, `/webtop`, or other Documentum directories. They will be lost when you upgrade the content of those directories. Place all of your custom content in the `/custom` (or user-defined) or `/WEB-INF` directories.

Components and actions are inherited in the same way that applications are inherited. Your application can extend the Webtop application definition, and within your application you can have a custom properties component that extends the webcomponent layer properties component and an objectlist component that extends the Webtop layer objectlist component. If your component or action extends a component that is not in the next layer below /custom you must make sure that your component or action is called and not a component or action in the intervening layer. For example, if your customized Webtop application will have a custom advanced search component, it should extend the advanced search component definition in the webtop layer, not the definition in the webcomponent layer. For more information on component inheritance, refer to [Component inheritance \(extends\)](#), page 224.

To create your application layer:

1. Use the application layer directory /custom, or rename the custom directory. If you rename the custom directory, be sure to specify the directory name in your /WEB-INF/web.xml file. (Refer to the last step in this set of steps.)
2. Add an app.xml file (refer to [Application configuration file \(app.xml\)](#), page 58) to the application layer directory and specify the name of the application that it extends. It must extend the top-level application layer, for example:

```
<application extends="wp/app.xml">
```

3. Create /config and /theme directories within your new application directory.
4. Create directories for your application-specific components and add the JSP pages and JavaScript files for those components.
5. Add configuration files for your components to the /config directory.
6. Add the supporting class files for your components to /WEB-INF/classes or, if they are archived in jar format, add them to /WEB-INF/lib.
7. (Optional) Specify your top-level application directory in the /WEB-INF/web.xml file. For example:

```
<context-param>
  <param-name>AppFolderName</param-name>
  <param-value>myapp</param-value>
</context-param>
```

Application environments

WDK-based applications can run in several types of environments with differing requirements:

- J2EE 1.3 Application servers

The standalone application environment supports Web applications in a J2EE application server. Authentication and configuration services are provided by the J2EE applications contained within the environment. Application state is maintained by binding with an HTTP session. Refer to *Web Development Kit Release Notes* for information on the supported application servers for the release that you have installed.

- JSR-168 compliant Portal servers

In portal environments, authentication and some preferences are controlled by the portal server. Portal Servers provide APIs with which a developer can create portlets that are aware of the portal server's environment and styles. Refer to *WDK for Portlets Release Notes* for information on the supported portal servers for the release that you have installed.

- Windows content authoring applications

Application Connectors provide support for access to WDK actions and components within a Windows authoring environment such as Microsoft Word.

Required application directories for custom applications

The following table lists the application layer directories required for a Web application based on the type of application you plan to build. For example, if you are building an application named `myapp` that uses WDK and its components but not Webtop, you first create a top-level directory named `/myapp`. Within that directory, you need the `/WEB-INF`, `/wdk`, `/webcomponent`, and `/custom` directories. (The `/custom` directory is not shown in the table below.)

Tip: All applications that contain customization or configuration require a separate directory to contain customizations. The `/custom` or user-defined directory is the top-level application layer.

The required directories must be located at the root of your Web application directory (for example, `/my_app/webcomponent`).

Table 2-1. Directories required for Web applications

Required directories						
Appli- cation based on	/WEB- INF	/wdk	/webcom- ponent	/webtop	/dam	/wp
WDK	X	X	X			
Webtop	X	X	X	X		
Portal app	X	X	X			
DAM	X	X	X	X	X	
DA	X	X	X	X		
Web Publisher	X	X	X	X	X	X

When you run the WDK installer and select **Create a new application**, you get the required directories for applications based on WDK and components. When you select **Customize an existing application**, the installer copies the existing WDK client application directories to a new location. For example, if you customize Webtop, you get WDK directories and the Webtop directory. In both types of installation, the installer also installs developer documentation, Javadocs, source code, native libraries and DFC jar files in the Documentum home directory on the J2EE server host.

Tip: To get the WDK component source code, run the WDK installer and select **Create a new application**. The source code is installed to the /webcomponent/src directory. The Webtop source code is installed to the /webtop/src directory.

Your components can extend WDK or Webtop components in the /custom directory. If you want to use a custom JSP page with an existing WDK component, you must extend the component in the custom layer and specify a different start page. Do not edit the WDK component definitions. Changes to installed JSP pages and XML files are not supported.

For information on customizing strings in WDK components, refer to [Configuring and localizing strings, page 137](#).

Web application archives (WAR files)

Your Documentum Web application can be packaged into a Web application archive (WAR) for deployment in a J2EE-compliant server. Your application must conform to the J2EE directory structure for Web applications as specified in the J2EE Servlet specification.

In order to carry out configurations, it is necessary to extract the files from the archive resource. The WDK installer expands the WDK WAR file when you select **Create a new application**. The WDK installer extracts the client application when you select **Customize an existing application**. Make a copy of the expanded WAR so that you always have a backup copy of the original contents of the installed WAR.

There are two types of files in the original WAR that you can edit:

- Externalized string files in /string directories and Java properties files in /WEB-INF/classes and its subdirectories. Be sure to back them up before you edit them.
- Branding themes, added to your application directory.



Caution: When you upgrade your existing application, WDK updates may overwrite customized properties or branding files. Back up your customized files.

For information on how to package a WAR for your custom Web application, refer to *Web Development Kit and Applications Installation Guide*.

How application elements interact

A WDK-based Web application consists of the WDK framework, WDK and custom components, controls, and actions. You may not need to customize all parts of the application in order to obtain the results that your business objective requires. The following topics describe how these various parts of the application interact.

- Controls

Controls represent a discrete UI functionality such as a button or link. They can be reused in many different JSP pages in the application and configured to perform a different function in each JSP page. The function that the control performs is based on a control event or action defined for the control. For example, a button may specify an onclick event handler that is handled in the current component. When the button is used in another component, the event handler may be named the same but perform differently.

- JSP pages

JSP pages are modeled as Form objects to handle HTML form state and browser history. A form is used within a component. A JSP page can contain several other JSP pages, each with a <dmf:form> tag. The parent JSP page must contain the <dmf:webform> tag to initiate form processing.

- Actions

Many controls in WDK JSP pages can call an action. WDK action classes evaluate preconditions and execute the action if preconditions are met. The execution usually launches a component to gather user input.

- **Components**
A component contains one or more forms and controls on the JSP pages, which make up the component UI. The component handles events raised by the component UI and updates the controls in the UI based on server processing or data binding.
- **Application layers**
The application layers in the application maintain application-wide functionality such as branding themes, accessibility settings, content transfer default settings, and supported locales.

Configuration overview

The WDK configuration model supports configuration of controls, components, actions, and applications. Controls are configured through JSP tags. Additionally, certain controls are configured through XML configuration files. Components, actions, and applications are configured through XML configuration files.

The following topics describe the general principles of configuration in WDK-based Web applications:

- [What is configurable, page 47](#)
- [Working with XML configuration files, page 49](#)
- [General configuration elements, page 50](#)
- [Extending XML definitions, page 51](#)
- [Scope, page 52](#)
- [Client environment qualifier, page 53](#)
- [Versioning, page 55](#)
- [Externalizing and configuring strings, page 56](#)

For information on specific types of configuration files, refer to the following topics:

- [Application configuration file \(app.xml\), page 58](#)
- [Action configuration file, page 215](#)
- [Component configuration file, page 221](#)
- [Attributelist configuration files, page 196](#)

What is configurable

The following parts of a WDK-based application can be configured:

- **Controls**

Controls render UI features such as buttons, tabs, HTML links. Controls are provided in a JSP tag library, which allows you to configure many aspects of the UI rendered as HTML. Basic controls, in the `dmf` tag library, provide standard Web functionality. Repository-enabled controls, in the `dmfx` tag library, provide data binding, validation, and formatting. You can configure controls through the JSP tag attributes on the JSP page itself and, for certain controls, through XML files. (Refer to [Controls that can be globally configured, page 175](#) for details on XML control configuration.)

- JSP pages (forms)

Forms are JSP pages that contain a `<dmf:webform>` or `<dmf:form>` tag. A form generates HTML form tags and maintains a model of the form state and browser history on the server. A form can include other forms, but generally there is a one-to-one correspondence between a form and a Web page. You can configure forms by changing the form layout in the JSP page itself.

- Actions

Actions associate UI events such as menu selection with application functions. Actions are usually launched by a UI element such as a link, button, list item, or menu item, or by a repository operation. An action consists of an action definition XML file and an action class that implements the action and determines whether a user can perform an action based on preconditions. The action control on a JSP page, such as a menu item or link, is enabled if the preconditions are met. You can configure actions in the action configuration file and configuring a control to launch an action.

- Components

Each component performs a specific repository task, such as check in or view renditions. A component can have the following resources: an XML file containing the definition of the component, a behavior class, an NLS resource file, a help file, and at least one JSP page (the layout page). You can configure components by changing the component definition or the JSP pages associated with it.

- Events

Events are raised when the user makes changes to elements in a UI form (JSP page). Events can be handled on the client, by JavaScript event handlers, or on the server, by the component class. You can configure events in the JSP pages by specifying the event handlers as control tag attributes and adding your custom client-side event handlers.

- Applications

Within a Web application, application layers are managed by the WDK framework. Application layer directories must be located in the root Web application directory. Application layers can inherit their configuration from a parent application layer. For example, the `webcomponent` layer inherits its application definition from the `wdk` layer. Configure an application in your custom configuration file `app.xml`, which is located in the top directory of your custom application layer. For example, if you are using the `/custom` layer, add your customization to `app.xml` in `/custom`.

- Branding

The branding service manages the UI look by themes, which incorporate images and icons, and cascading style sheets (CSS). You can apply styles at any level of granularity: on an individual control, on a component, on a group of components within a container, and on the entire application. You can configure branding through themes, and you register your brand in the application configuration file `app.xml`. (Refer to [Branding an application, page 122.](#)) Users select a theme for display in the Preferences component.

- Text strings

UI Strings and error messages are externalized into Java `*.properties` files. These text files allow you to change or localize the text of buttons, links, labels, and messages without any knowledge of Java (refer to [Configuring and localizing strings, page 137](#) and [Externalizing and configuring strings, page 56](#)). WDK supports localization (translation) of the UI strings through national language support (NLS) lookup. Locales are specified in the application configuration file `app.xml`. The localized strings are locale-specific. The application uses the string for the user's selected locale.



Caution: When you change WDK files, all of your changed files except for `*.properties` files in `/WEB-INF/classes` and its subdirectories should be in a custom folder that will not be overwritten by product upgrades. You must back up all customized files before upgrading an existing Web application. After backup and installation of a newer version of WDK, compare your backed-up files to the newer versions that were installed. You might need to change some of your customized properties files and JSP pages due to changes in the supporting Java classes that use these files.

Working with XML configuration files

Actions, components, and the application itself are configured through XML configuration files.

Configuration files utilize the WDK configuration service, which retrieves values in a context-sensitive manner. The configuration framework contains National Language Service (NLS) functionality for looking up localized string tag values. The `<nlsid>` and `<nlsbundle>` elements allow strings to be specified in a language-independent manner within configuration files and component JSP pages. For more information on string configuration, refer to [Configuring and localizing strings, page 137](#).

Note: J2EE servers do not recognize changes to XML files automatically. Therefore, for your changes to take effect you must either restart the server or refresh the component definitions by navigating to the refresh utility page `/wdk/refresh.jsp`.

A WDK-based application such as Webtop or Web Publisher is contained within a single root directory. This single directory contains application layer directories: `/wdk`,

/webcomponent, application-specific directories, and /custom. Each application layer is configured in an XML file named app.xml. Each application directory contains a /config directory that contains all of the configuration files for the application layer's components and actions.

General configuration elements

The following elements are present in all WDK configuration files:

```
1<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
2<config version="1.0">
3<scope qualifier_name="qualifier_value">
4  <primary-element
5    extends="primary-element:path_to_definition_file
6    notdefined="true_or_false"
7    version="version_number">
8    <nlsbundle>fully_qualified_bundle_name</nlsbundle>
9    <filter qualifier_name="qualifier_value">
10   <filtered_element>element_value</filtered_element></filter>
   </primary-element>
</scope>
</config>
```

- 1 Configuration files are written in XML and begin with the standard XML language declaration.
- 2 The root element of the configuration file is the <config> element. The entire configuration definition is contained within this element. The version attribute is reserved for future changes to the WDK configuration framework. For WDK 5.3, the config version attribute value is 1.0.
- 3 Defines the context in which the configuration definition applies. The context is matched to a qualifier value. If no qualifier is specified, the definition applies to all contexts that are not otherwise defined. Refer to [Scope, page 52](#) for more information on scope.
- 4 One or more elements that represent a definition, such as <action>, <component>, <application>, <attributelist>, and <docbaseobjectconfig>. You can specify more than one primary element within a configuration file. Each type of primary element (application, component, or action) has additional elements that are required by the particular kind of primary element. The elements common to all primary-element definitions are described below.
- 5 Specifies a definition that is inherited by the current definition. Elements that are not named in the current definition are inherited. Elements that are named override the parent definition. If you wish to include certain child elements of the element that you

override, you must copy those elements into your definition. Refer to [Extending XML definitions, page 51](#) for more information.

- 6 Configurations can hide elements that are defined in the parent scope by using the `notdefined` attribute. Refer to [Hiding components, page 227](#) for more information. In the following example, the checkout component is not available for folders:

```
<scope type='dm_folder'>
  <component id="checkout" notdefined="true"></component>
</scope>
```

- 7 Specifies a version number for the component or action. If the version attribute is not present or has a value of `CURRENT`, the component is assumed to be the latest version. For more information, refer to [Versioning, page 55](#).
- 8 Specifies the fully qualified name of an NLS resource bundle that provides localizable strings for the component. The bundle consists of a properties file with the bundle name and the `.properties` extension. Each application layer can override the content of an NLS property file in the base applications. For more information on string configuration, refer to [Configuring and localizing strings, page 137](#) for more information on string configuration.
- 9 Optional filter that applies a scope qualifier such as type or role, which limits the application of contained elements to user contexts that match the scope value. Refer to [Hiding component features, page 228](#) for more information.
- 10 Element to which the filter is applied, for example, `<enableshowall>`

Extending XML definitions

When you customize an existing WDK component or action, you must extend its configuration definition. For example, your custom properties component can extend the Webtop properties component and use a custom JSP page to change the UI. The elements and values in the Webtop configuration definition are inherited by your component definition unless you override elements in the definition. You can also add elements or parameters to the definition.

Make sure that you extend the definition from the highest layer in your application. This will ensure that your definition inherits all of the current functionality for that feature. For example, your custom advanced search component should extend the Webtop advanced search definition rather than the webcomponent layer definition.

Create your extended definition in the `/custom/config` directory or subdirectories. Do not create backup copies of definitions in the `/config` directory or subdirectories, because the configuration service will throw a duplicate element exception.



Caution: When you extend an XML definition, you do not need to copy the entire contents of the base definition. However, if you define an element, you must copy all of the contents of that element if you wish them to be a part of your definition. For example, if you extend the WDK doclist component and your definition contains an <objectfilters> element, you must copy all of the <objectfilter> elements and their contents if you wish them to be in your extended component.

To extend a definition, the primary element in the definition, such as <component> or <application>, must use the extends attribute. The primary element can extend a file or a component and can specify a particular scope within the file or component. In the first example below, <application> is the primary element. The custom application definition extends the webcomponent layer configuration file in the following example:

```
<application extends="webcomponent/app.xml">
```

The extended configuration is specified in one of the following ways:

- The name of the configuration file that is extended. (The extended primary element must be unique within the referenced file or scope. For example, if the configuration file contains two component definitions, you must also specify the primary element value that is extended.) In the following example, the custom component extends a file that contains a single component definition:

```
<component id="my_drilldown" extends=
  "webcomponent/config/navigation/drilldown/drilldown_component.xml"
```

- The primary element value and file name that is extended. This method is preferred over the previous method, because it always returns the correct definition in the configuration file. (Configuration files can contain more than one definition.) In the following example, the custom component extends a component within a configuration file that contains more than one definition:

```
<component id="my_container" extends=
  "locatorcontainer:webcomponent/config/library/locator/
  locatorcontainer_component.xml">
```

- File name or primary element name plus scope within the referenced definition. In the following example, a custom action extends a scoped action definition in the webcomponent application layer:

```
<action id="my_rendition" extends=
  "importrendition:application='webcomponent'
  type='dm_sysobject'"/>
```

Scope

The configuration service uses the user's context, such as selected object, user role, or current component, to resolve the appropriate scope and deliver the component that is defined for the scope. Scope is defined by a qualifier class that implements the IQualifier

interface. WDK includes the following qualifiers: Documentum type, role, privilege, client environment, and application. Webtop adds qualifiers for repository name and location in the browser tree. You can add custom qualifiers to your Web application.

The configuration service matches the user's context to the closest matching qualifier value. For example, if the user has selected a folder to view its attributes, the configuration service finds the definition for the attributes component that is scoped to the type `dm_folder`. The component definition for `dm_folder` scope specifies a different UI from the component definition for `dm_sysobject` scope.

You can use scope in the following ways:

- An action or component definition can inherit or override scope

The definition for a type inherits the defined parameters and configurable elements for the parent definition unless they are specifically overridden. To override an element, you must provide different content for the element. For example, to require a different set of parameters for the child type, define a `<params>` element in the component or action definition that contains your new `<param>` elements. If your extended definition has no `<params>` element, all parameters are inherited.
- A qualifier can have more than one value

The list of valid values is comma-separated. For example, `<scope type='dm_document, dm_folder'>` applies to both types of objects and types descended from these types.
- An action or component definition can have more than one qualifier

For example, the `newgroup` action is scoped to `type='dm_group'` and `privilege='creategroup'`.
- An action or component definition can exclude qualifier values using the NOT operator

For example, a definition can be scoped to `type='dm_group' privilege='not createtype'`. As a result, any user could create a new group unless the user had the privilege `createtype`. In another example, a properties definition that is scoped to `role='contributor, not administrator'` would have no definition for the administrator role (no access to the feature).

Scope is resolved in the order that qualifiers are specified in `app.xml`.

Client environment qualifier

The `clientenv` qualifier is introduced to enable component configuration based on the client environment. This qualifier matches the context value `"webbrowser"` or `"appintg"` to the context `"clientenv"`.

The client environment context is established in one of the following ways:

- Specify `clientenv` in a URL request parameter, preferably in the first URL request that invokes a WDK-based application. This method takes precedence over the value in `app.xml` (the second option). For example:
`http://webtop/component/main?clientenv=webbrowser`
- Specify the default `clientenv` value in your custom `app.xml` as the value of `<application>.<environment>.<clientenv>`. Valid values: `webbrowser` | `appintg` | `not appintg`

The client environment is stored as a session cookie so that WDK can restore the context for a given client session when the server times out.

The `clientenv` qualifier can be used as a filter in component definitions to specify a different set of JSP pages depending on the environment. For example, the `changepassword` component definition uses the value of `appintg` and `not appintg` to present different pages depending on whether the user is in an Application Connectors environment. The filter is applied as follows:

```
<pages>
  <filter clientenv="not appintg">
    <start>/wdk/system/changepassword/changepassword.jsp</start>
  </filter>
  <filter clientenv="appintg">
    <start>/wdk/system/changepassword/appintgchangepassword.jsp</start>
  </filter>
</pages>
```

A more granular environment configuration can be applied within a JSP page itself using the `clientenvpanel` control. This control is used to show or hide UI elements based on the runtime `clientenv` context. The client environment is specified as the value of the `environment` attribute. For example, the 5.3 `checkin` component JSP page has a `clientenvpanel` control that is rendered only in the `AppConnectors` environment:

```
<dmfx:clientenvpanel environment="appintg">
  <dmf:fireclientevent event="aiEvent"
    includeargname="true">
    <dmf:argument name="event" value="ShowDialog"/>
    ...
  </dmf:fireclientevent>
</dmfx:clientenvpanel>
```

The `reversevisible` attribute on the `clientenvpanel` control toggles the display. In the same example above (`checkin.jsp`), another panel is hidden in the `AppConnectors` environment:

```
<dmfx:clientenvpanel environment="appintg" reversevisible="true">
  ...
  <dmf:checkbox name="checkinfromfile"
    id="checkinfromfile" nlsid="MSG_CHECKIN_FROM_FILE"
    onclick="onCheckinFromFileClick" runatclient="true"/>
  ...
</dmfx:clientenvpanel>
```

JSP fragments can also be defined, so that a component JSP page includes fragments that are dispatched based on the client environment. For information on JSP fragments, refer to [JSP fragment control](#), page 189.

To trace problems in the client environment, turn on the tracing flag CLIENTENV.

Versioning

Component and action definitions, or any other configuration definition within a <config> element, can have more than one version. The version is denoted by a version attribute on the scope element in the definition, for example, <scope version="5.2.5">.

Supported versions are registered in /wdk/app.xml as the values of <supported_versions>.<version>. The WDK current version is supported and does not need to be registered in app.xml. The current component or action definition is dispatched unless there is a older version for the same action or component ID in the /custom/config directory with the same component ID. For example, if your custom search component extends the WDK 5.2.5 search component and has the same component ID, your custom component will be launched in place of the newer WDK 5.3 search component. (A version attribute value of 5.2.5 has been added to all WDK 5.2.5 component definitions that are superseded by 5.3 components.)

The following table shows the supported and unsupported configurations of versioned components and containers:

Table 2-2. Binding scenarios for versioned components

Custom layer	WDK layer	Binding
Extends 5.2.5 container of same ID	5.2.5 component loaded by 5.2.5 container (refer to note below)	Succeeds
Extends 5.2.5 component	5.2.5 container loaded by 5.2.5 action	Succeeds
Extends 5.2.5 container and component of same IDs	(refer to note below)	Succeeds

Custom layer	WDK layer	Binding
Extends 5.3 container and contains 5.2.5 component		Fails. Must extend 5.2.5 container to use 5.2.5 component or extend 5.3 component to use 5.3 container
Extends 5.2.5 container and contains 5.3 component		Fails. Must extend 5.2.5 component to use 5.2.5 container or extend 5.3 component to use 5.3 container.

Note: If your custom container extends a 5.2.5 container but has a different component ID, your container definition must have a `<bindingcomponentversion>` element with a value of 5.2.5.

The version number has the form `n.n.n.n` where `n` is an integer. You can version your own components, using the WDK inheritance procedure. Register your version numbers in the `<supported_versions>` element of `/custom/app.xml`. Versions will take precedence in the order that they are listed in this element: The first version in the list has the highest precedence.

Externalizing and configuring strings

UI text and error messages in WDK controls, actions, and components are externalized in Java properties files. Properties files are text-based files that are used by Java classes for initialization settings. This externalization of strings facilitates the testing and translation of strings.

The following kinds of properties files are used in WDK:

- UI strings, which are externalized to properties files for translation (refer to [Configuring and localizing strings, page 137](#))

Strings are externalized from each application layer into the `/strings` subdirectories.

- Tracing and logger initialization files ([Tracing, page 333](#) and [Logging, page 343](#))

Tracing and logging settings assist you in debugging your application.

- Help location file

The file `/WEB-INF/com/documentum/web/common/HelpService.properties` establishes the location of help files in the application. For more information on help files support, refer to [Help service, page 590](#).

- Accessibility image strings ([Image accessibility strings, page 587](#))

Strings are displayed as alt text for images in Web applications. These strings are located in each application layer in the subdirectories of `/strings/com/documentum/layer_name/accessibility`.

- Form processor settings

Settings for browser history, processor navigation hooks, timeout URL, history released URL, no return URL, server busy URL, event handler timeout, and history size are configured in `FormProcessorProp.properties` located in `/WEB-INF/classes/com/documentum/web/form`. Refer to [Navigation defaults, page 97](#) for more information on these settings. Additionally, you can set the form tag attribute `keepfresh` to `true` to force a reload when the user navigates using **Back** and **Forward** buttons.

- JavaScript files

The list of JavaScript file references that are generated with every parent JSP page containing the tag `<dmf:webform>` are configured in `WebformScripts.properties` located in `/WEB-INF/classes/com/documentum/web/form`. Refer to [Registered scripts, page 115](#) for more information.

- Cache size for databound queries

The cache size is configured as the `cachesize` setting in the file `DataboundProperties.properties` located in `/WEB-INF/classes/com/documentum/web/form/control`. The query will be reused when the user pages beyond the cached results.

- J2EE principal

The trusted authenticator credentials are configured in `TrustedAuthenticatorCredentials.properties` located in `/WEB-INF/classes/com/documentum/web/formext/session`. Refer to [J2EE principal authentication, page 104](#) for more information.

Configuring an application

You can configure the following resources in a WDK-based application:

- [Application name, page 58](#)
- [Application configuration file \(app.xml\), page 58](#)
- [Web deployment descriptor \(web.xml\), page 83](#)
- [Application environment properties, page 88](#)
- [Configuring application failover support, page 89](#)
- [Configuring content transfer mode for the application, page 91](#)
- [Virtual links, page 91](#)
- [Content server event notification, page 97](#)

- [Navigation defaults, page 97](#)
- [Browser history, page 98](#)
- [Cookies, page 99](#)
- [Timeout, page 101](#)

Application name

The top application layer by default is the custom application. Specify the top application layer in the web.xml file, located in /WEB-INF. The application layer context parameter name is AppFolderName, the value of the element <param-name>. Specify the name of your custom application base folder as the value of the element <param-value>. For example, Webtop specifies custom as the value of <param-value> for the <param-name> AppFolderName.

The value of AppFolderName is used by the configuration service to determine application definition inheritance. If your top application layer folder is named something other than custom, you must change the value in web.xml to match the folder name.

Application configuration file (app.xml)

You can configure application-wide behavior through the application configuration file app.xml. Each application layer, such as wdk, webcomponent, webtop, or wp, has an app.xml file. The file contains an application definition within the elements <application></application>.

To inherit and override settings in another application layer, your application definition can extend an application definition in another layer. For example, the webcomponent app.xml file specifies the inheritance in the application element as follows:

```
<application extends="wdk/app.xml">
```

For more information on application inheritance, refer to [Application layer inheritance, page 42](#).

The WDK application configuration files, /wdk/app.xml and /webcomponent/app.xml, contain the following elements. Each set of elements is described in the sections that follow this XML file transcription.

```
<config> See <config> element, page 60
<scope> See <scope> element, page 60
<application> See <application> element, page 60
    <qualifiers> See <qualifiers> element, page 60
```

`<environment>` See `<environment>` element , page 60

`<failover>` See `<failover>` element, page 62

`<fallback_identity>` See `<fallback_identity>` element, page 62

`<language>` See `<language>` element, page 62

`<save_credential>` See `<save_credential>` element, page 63

`<authentication>` See `<authentication>` element, page 63

`<rolemodel>` See `<rolemodel>` element, page 64

`<themes>` See `<themes>` element, page 64

`<accessibility>` See `<accessibility>` element, page 65

`<contentxfer>` See `<contentxfer>` elements, page 66

`<browserrequirements>` See `<browserrequirements>` element, page 70

`<errormessageservice>` See `<errormessageservice>` element, page 70

`<infomessageservice>` See `<infomessageservice>` element, page 70

`<requestvalidation>` See `<requestvalidation>` element, page 71

`<adobe_comment_connector>` See `<adobe_comment_connector>` element, page 72

`<notification>` See `<notification>` element, page 72

`<session_config>` See `<session_config>` element, page 73

`<xmlfile_extensions>` See `<xmlfile_extensions>` element, page 73

`<formats>` See `<formats>` element, page 74

`<preferred_renditions>` See `<preferred_renditions>` element, page 75

`<modified_vdm_nodes>` See `<modified_vdm_nodes>` element, page 76

`<custom_attribute_data_handlers>` See `<custom_attribute_data_handlers>` element, page 77

`<discussion>` See `<discussion>` element, page 77

`<xforms>` See `<xforms>` element, page 77

`<listeners>` See `<listeners>` element, page 78

`<client-sessionstate>` See `<client-sessionstate>` element, page 78

`<dragdrop>` See `<dragdrop>` element, page 79

`<copy_operation>` See `<copy_operation>` elements, page 79

`<move_operation>` See `<move_operation>` elements, page 80

`<richtexteditor>` See `<richtexteditor>` element, page 80

`<plugins>` See `<plugins>` element, page 80

`<display>` See `<display>` element, page 81

`<job-execution>` See `<job-execution>` element, page 82

`<config>` element

Root element for all WDK configuration files

`<scope>` element

Limits the configuration to contexts that match the scope attributes. The scope is not defined in this example here because app.xml files are unqualified, that is, they apply to the entire application.

`<application>` element

Contains all elements that configure application-wide behavior

`<qualifiers>` element

Contains `<qualifier>` elements that define scope for configuration files in the application. Each `<qualifier>` element contains the fully qualified class name of a class that implements `IQualifier`.

`<environment>` element

Sets environment-specific dispatching of components and other environment-specific configuration. The `<environment>` element contains one `<clientenv>`, one `<clientenv_structure>`, and one `<serverenv>` element.

Table 2-3. Client environment elements (<clientenv> and <clientenv_structure>)

Element	Description
<clientenv>	Specifies the applicable client environment. Valid values: webbrowser portal appintg * (all client environments). Default = webbrowser. Values can be used to scope action or component definitions or filter definition elements.
<clientenv_structure>	Defines the branches of the client environment specified in <clientenv>. Contains one or more <branch> elements.
.<branch>	Contains a <parent> and a <children> element defining the branch for the specified client environment
.<parent>	Names a client environment branch. Values can be used to scope action or component definitions or filter definition elements.
.<children>	Specifies child environments of the parent. Contains one or more <child> elements.
.<child>	Specifies a client environment child of the parent environment. Values can be used to scope action or component definitions or filter definition elements.

Table 2-4. Server environment elements (<serverenv>)

Element	Description
<filter>	The value of the clientenv attribute must match one of the client environments defined in <environment>.<clientenv>.
.<class>	Fully qualified name of a class that instantiates the server environment
.<preferencestoreclass>	Fully qualified name of a class that instantiates the server environment

<failover> element

Enables application failover in <failover>.<enabled>, which turns on serialization for all failover-enabled components and sessions in the application. By default, failover is enabled. If you migrate WDK 5.2.5 customizations to WDK 5.3, and your custom classes do not support failover, disable failover in your custom app.xml file.

For more information on failover support, refer to [Configuring application failover support, page 89](#).

<fallback_identity> element

Turns on the DFC fallback identity feature `IDfSessionManager.setIdentity(*)`. This flag is enabled by default. With the introduction of foreign objects in the 5.3 release, some WDK 5.2.5 components or your custom component that call `getSession()` will cause the component to attempt to get a session for every possible repository in the repository list. In this case, you should turn off the fallback identity flag. If you have extended the 5.2.5 Webtop browsertree component, for example, you must turn off the fallback identity flag.

<language> element

Contains elements that set the supported locales, default locale, and fallback language.

Table 2-5. Language elements (<language>)

Element	Description
<supported_locales>	Contains a <locale> element for each supported locale for which there are localized strings in the application
<locale>	Java locale names are constructed from a concatenation of the two-letter ISO language code and the two-letter ISO country code in the form <code>xx_YY</code> , where <code>xx</code> is the two-character lower-case language code and <code>YY</code> is the two-character uppercase code. The language code alone (<code>YY</code>) is an acceptable locale code string.

Element	Description
<default_locale>	Specifies the locale to be shown before a user selects the preferred locale.
<fallback_to_english_locale>	Specifies whether wdk will fall back to use the English (US) locale string if a resource string is not available for a specified locale. For development, you may wish to set this to false to identify non-localized strings, which will be displayed as xxNLS_IDxx where NLS_ID is the NLS key that does not have a value in the specified locale.

<save_credential> element

Contains elements that are used for saving user's credentials (username and password) for a repository.

Table 2-6. Save credentials elements (<save_credentials>)

Element	Description
<enabled>	Set to true to enable saved credentials
<encryption_key>	Specifies a string encryption key. Must be identical across all WDK application instances on the application server. If the key is changed by an administrator, users will be prompted for login. You can use the trusted authenticator tool (com.documentum.web.formext.session.TrustedAuthenticatorTool).
<disabled_docbases>	Specifies repositories that will not support saved credentials

<authentication> element

Contains elements whose values are used for login: domain, repository, authentication service class, and single sign-on.

Table 2-7. Authentication elements (<authentication>)

Element	Description
<docbase>	Default repository name. When SSO authentication is enabled but a repository name is not explicitly spelled out by the user nor defined in this element, the sso_login component is called. In this case the component will prompt the user for the repository name.
<domain>	Windows network domain name
<service_class>	Fully qualified name of class that provides authentication service. This class can perform pre- or post-processing of authentication.
<sso_config>	Single sign-on authentication configuration elements
<sso_config>. <ecs_plug_in>	Name of the Content Server authentication plugin (not the authentication scheme name). Valid values: dm_netegrity dm_rsa
<sso_config>. <ticket_cookie>	Name of vendor-specific cookie that holds the sign-on ticket, for example, SMSESSION (Netegrity)
<sso_config>. <user_header>	Name of vendor-specific header that holds the user name. The user_header value is dependent on the settings in the webagent configuration object in the policy server. The default is either SMUSER or SM-USER depending on whether the flag "LegacyVariable" is set to true or false. If false, it uses SMUSER, if true, it uses SM-USER.

<rolemodel> element

Configures the class that defines roles used by the role qualifier

<themes> element

Contains elements that define the themes available in the preferences UI.

Table 2-8. Theme elements (<themes>)

Element	Description
<default_theme>	Specifies the default theme to be used when the application starts up.
<theme>	Lists all of the themes available in the application.
<theme>.<name>	Name of a theme.
<theme>.<label>	NLS key in the branding properties file that provides a label for the theme.
<nlsbundle>	Specifies the NLS resource file that contains a localizable list of themes for the application. This list will be displayed for user preference selection.

<accessibility> element

Contains elements that turn on accessibility support (section 508 compliance) in the application.

Table 2-9. Accessibility elements (<accessibility>)

Element	Description
<accessibility>	Contains settings that turn on accessibility support.
<alttextenabled>	Flag to enable alt text. If true, alt text will be displayed for all icons and images.
<keyboardnavigationenabled>	Flag that enables keyboard navigation through menus, buttons, and tabs via the keyboard tab and arrow keys
<shortcutnavigationenabled>	Flag that generates a shortcut to the top and bottom of a tree

<contentxfer> elements

The elements within <contentxfer> set the application-wide content transfer mechanism. The elements within <contentxfer>.<common> turn on or off compression for WDK 5.2.5 applet (not UCF) content transfer operations.

Note: Most of the settings in <contentxfer> are used for WDK 5.2.5 applets only, except where noted. For information on the UCF settings, refer to [Configuring the UCF application server](#), page 521.

Table 2-10. Content transfer common elements

Element	Description
<default-mechanism>	Sets the content transfer mode for the application. Valid values: http ucf. If one or more custom components extend pre-5.3 applet components, specify ucf. The UCF mechanism is also required to support Adobe Comment Connector.
<common>	Contains 5.2.5 configuration elements for applet content transfer
.<inlinecompressionwindows>	Deprecated: for 5.2.5 custom content transfer only. Set to true to compress content transfer on Windows clients. If most objects transferred are already compressed, such as MPG or JPG, set to false for better performance.
.<inlinecompressionmacosx>	Deprecated: for 5.2.5 custom content transfer only. Mac OS X client content transfer compression (refer to above)
.<inlinecompressionmacos9>	Deprecated: for 5.2.5 custom content transfer only. Mac OS 9 client content transfer compression (refer to above)

Table 2-11. Content transfer ACS elements

Element	Description
<acs>	Contains elements that configure Accelerated Content Services: <enable>, <attemptsurrogateget>, <maintainvirtuallinks>

Element	Description
.<enable>	Set to true to enable Accelerated Content Services in the application. Requires configuration of Accelerated Content Services on the network.
.<attemptsurrogateget>	Set to false to display only ACS servers that require surrogate get, which fetches from a remote storage area to the storage area local to the selected Accelerated Content Services server. Set to true to display all ACS servers.
.<maintainvirtuallinks>	To maintain relative links within HTTP content transfer, set to true. If true, ACS will not be used for HTTP view operations. If false, ACS will be attempted for view operations, which will result in relative links inside the viewed document being broken in the browser. Checkout, export, and edit operations in HTTP mode are not be affected by this flag, since they do not display content inline in the browser.

Note: WDK sets the ACS transfer protocols for HTTP-based transfer to HTTP, HTTPS for HTTP sessions and to HTTPS for HTTPS sessions. For UCF-based transfer, WDK does not set any protocol. The transfer is based on ACS configuration only.

<contentxfer>.<client> elements specify default locations for applet content transfer on the client (WDK application users).

Note: Most of the content transfer settings in app.xml do not apply to UCF. The settings that do apply to UCF, in <contentxfer>.<server>, are so noted in the table below. For information on UCF client configuration, refer to [Configuring the UCF application server, page 521](#).

Table 2-12. Client applet content transfer elements (<contentxfer>. <client>)

Element	Description
<contentlocationunix>	Not used for UCF or HTTP. Path to default location for content on UNIX and Mac clients. Send and receive directories will be created under this location the first time content transfer takes place. The directories created will be relative to the user's home directory (returned by the "user.home" java system property).

Element	Description
<contentlocationwindows>	Not used for UCF or HTTP. Path to default location for content on Windows clients. Send and receive directories will be created under this location the first time content transfer takes place, if a value for this location is not already set in the Windows registry.
<checkoutlocationunix>	Not used for UCF or HTTP. Path to default location for checkout on UNIX and Mac clients. Send and receive directories will be created under this location the first time content transfer takes place.
<checkoutlocationwindows>	Not used for UCF or HTTP. Path to default location for checkout on Windows clients. Send and receive directories will be created under this location the first time content transfer takes place.
<userlocationunix>	Not used for UCF or HTTP. Path to default location for files being viewed.
<userlocationwindows>	Not used for UCF or HTTP. Path to default location for files being viewed.
<registrylocationunix>	Not used for UCF or HTTP. Path to default location for the WDK registry file on UNIX and Mac clients. WDK applications track information about checked out files and their locations in the WDK registry file.
<buffersize>	Not used for UCF or HTTP. Size of content transfer memory buffer on the HTTP server.
<debug>	Not used for UCF or HTTP. Turns on client content transfer debugging (temporary files are not removed). You can determine how far transfer progressed. For example, during upload, the content is sent from the client to a temporary directory on the server (<server>.<contentlocationwindows>). If content does not successfully transfer from the repository to the WDK server, you will see the temporary upload in this directory. Requires restarting the JSP server.

Element	Description
<uploadchunksizeplatform_name>	Not used for UCF. Size in bytes of each chunk that is uploaded by the client for content transfer (for example, for import or checkin). Size can differ for Windows and Macintosh clients.
<housekeepinginterval>	Not used for UCF or HTTP transfer. (The registry key is read by UCF operations in DFC, but the setting in app.xml is not read.) Interval in days between download and cleanup of temporarily viewed files on the client. Cleanup takes place at next login after the interval has expired. This setting is overridden by the registry key HKEY_CURRENT_USER\Software\Documentum\Housekeeping\NumberOfDays.

<contentxfer>.<server> elements specify default locations for content transfer files on the application server file system. Settings that do not apply to UCF can be configured in ucf.server.config.xml, located in /WEB-INF/classes.

Table 2-13. Server content transfer elements (<contentxfer>.<server>)

Element	Description
<contentlocation>	Not used for UCF. Relative path to default location for temporary content on the HTTP server. The root for this relative path is the web application root. Send and receive directories for content transfers through the server will be created under this location the first time content transfer takes place. Default: WEB-INF/contentXfer
<bufferize>	Not used for UCF or HTTP. Size of content transfer memory buffer on the web client. Default: 4096
<debug>	Not used for UCF. Refer to <client>.<debug> description. Turn on both client and server debugging to determine where content transfer is failing.
<version>	Not used for UCF or HTTP. Sets the version of the content transfer applets used by the application

<browserrequirements> element

The elements within <browserrequirements> enforce supported browsers and platforms and provide strings for error messages.

Table 2-14. Browser requirement elements (<browserrequirements>)

Element	Description
<windows>	Contains the list of supported browsers for the Windows platform
<macintosh>	Contains the list of supported browsers for the Macintosh platform
<unix>	Contains the list of supported browsers for the Unix platform
<nlsbundle>	Contains the localized error messages for the browserrequirements control
<warningmessage>	Contains the warning message or NLS ID of the warning message
<unsupportedplatformmsg>	Contains the error message to be displayed when the client is on an unsupported platform
<unsupportedbrowsermsg>	Contains the error message to be displayed when the client is using an unsupported browser
<javadisabledmsg>	Contains an error message to be displayed when Java is disabled in the browser
<browserversionmsg>	Contains an error message to be displayed when the browser is not a supported version
<softwareinstallmsg>	JavaScript handler not currently implemented

<errormessageservice> element

Contains a class element for a class that provides error messages.

<infomessageservice> element

Contains a class element for a class that provides informative messages

<requestvalidation> element

Turns on validation for each HTTP request to detect possible malicious scripting (cross-site scripting). Each possible URL request parameter can be configured here to be tested for conformity to a datatype and a regular expression.

Note: You can also turn off stack trace display for production applications in the `errormessage` component definition. This will prevent attackers from gaining information about the application.

Table 2-15. URL request validation elements (<requestvalidation>)

Element	Description
<enabled>	Set to true to validate every HTTP request for the parameters named in the <parameter> elements.
<parameter>	Contains a parameter that must be validated. Must contain a <name> element and at least one of the following child elements: <datatype>, <regexp>, <validator>. For each URL parameter, WDK checks for a parameter named in this configuration list. It creates a validator if one is specified, or a regular expression validator if regexp validation is specified for the parameter. Then it checks the datatype if one is specified.
<name>	Required. Must contain a valid URL request parameter.
<validator>	Fully qualified class name for a validator to validate the parameter value. Must implement <code>IRequestParamValidator</code> .
<regexp>	Regular expression that must be satisfied by the parameter value, for example, <code>__client(\d+)(~)(\d+)</code> . For information about Apache expression syntax, refer to the Apache Web site .
datatype	Datatype of the parameter value, for example, String, Integer, Long. String type requires a regular expression (<regexp>).

Note: The URL parameters that are configured in `app.xml` represent parameters that are added by the Form class and by several other classes that append these parameters to a URL. For a description of some of these parameters, refer to the javadocs for `IParams`. The `__dmfHiddenX` and `Y` parameters are hidden parameters that are used to save

browser scroll offsets in order to refresh a display. The `__dmfSerialNumber` parameter is set by the Form class to aid in automated testing.

<adobe_comment_connector> element

(In `/webcomponent/app.xml`) The following elements set connection and formats for the Adobe comment connector servlet. Requires Adobe Acrobat 6, PDF Annotation Services, and UCF content transfer (`<default-mechanism>` element value in `app.xml`).

Table 2-16. PDF Annotation Services elements (<adobe_comment_connector>)

Element	Description
<code><server_url></code>	Specifies the base URL to the Adobe comments servlet, for example, <code>http://myserver:port/</code>
<code><use_virtual_link></code>	Set to true to use a virtual link to retrieve content. Set to false to enable comment on the current document only.
<code><formats></code>	Contains all of the formats that can support Adobe comments
<code><format></code>	Specifies a format for which to allow Adobe comments, for example, pdf or msw8.

<notification> element

The following elements set the Content Server events on a subscribed document for which a user can turn on notification. Notification on replica and reference (shortcut) objects is not supported. For more information on event notification, refer to [Content server event notification, page 97](#).

Table 2-17. Event notification elements (<notification>)

Element	Description
<events>	Contains one or more <event> elements that will be made available to users who are selecting notification
<event>	Specifies the name of a Content Server API event

<session_config> element

Contains session management settings:

Table 2-18. Session management elements (<session_config>)

<max_sessions>	Sets the maximum number (integer) of application server sessions. After the maximum number of sessions has been reached, requests are redirected to the JSP page /wdk/serverBusy.jsp. A value of -1 means that there is no limit to the number of sessions.
<timeout_control>	Forces timeout of HTTP session
<client_shutdown_session_timeout>	Specifies the number of seconds before the session will be shut down after the main frame has been unloaded by user action. Default = 120 seconds if no configuration element is present, minimum = 15 seconds. If the timeout is larger than the actual HTTP session timeout configured in web.xml, the session timeout will not be overridden.

<xmlfile_extensions> element

Contains <extension> elements, which configure the file extensions that will be recognized and parsed as XML files by the application. You must configure these extensions if your application connects to a version 5.1 repository. If your application connects to a 5.2 repository or later, the WDK-based application will retrieve the

dm_format object based on the file extension and use the object's format_class attribute to determine whether the file extension indicates an XML format file. If no dm_format object is found for the file extension, the extensions in <xmlfile_extensions> will be used.

Table 2-19. XML extensions elements (<xmlfile_extensions>)

Element	Description
<extension>	Specifies a file extension that will be parsed as an XML file, for example, xsl, xml, and txt.

<formats> element

Contains a list of file extensions that map to known formats in the repository, mapping for format extensions on specific platforms, and an optional class that performs extensions mapping.

Table 2-20. Formats elements (<formats>)

Element	Description
<custom-file-extensions>	Contains one or more <format> elements that map a custom file extension to a format in the repository
<format>	Specifies a file extension as the value of extension attribute. Specifies the name of a format in the repository as the value of the name attribute.
<extension-format-detection>	Contains optional <class>, <client>, and <default> elements.
<class>	Specifies the fully qualified class name for a custom file extension detection and mapping class that implements java.util.Map. Overrides the settings in <extension-format-detection>.

Element	Description
<client>	Contains one or more <format> elements that override the format mapping in the <default> element. The platform attribute specifies the platform for which the mapping is defined. Valid values for platform are defined as static variables of the WDK ClientInfo class. For example, in the /wdk/app.xml configuration, the extension txt is mapped to the text format for browsers on the UNIX platform, which overrides the default cttext format. Clients should be listed in order of more specific to less specific. For example, the Mac OSX is one of the UNIX platforms, so Mac OSX mappings would be listed before UNIX mappings.
<default>	Contains <format> elements that map a file extension to a format in the repository

<preferred_renditions> element

Contains elements that specifies the application to be used for viewing or editing a specific document type and format combination. Users can override these settings using the preferred renditions component.

Table 2-21. Preferred renditions elements (<applications> and <renditions>)

Element	Description
<renditions>	Specifies the default list of renditions (document type and format combinations) and the application to be used for viewing or editing
.<rendition>.<mode>	Action to be performed. Valid values: view edit
.<rendition>.<objecttype>	Object type in the repository, such as dm_document, or all_types

Element	Description
.<rendition>.<primaryformat>	The primary format to be used for editing objects of the specified type. May not be the same as the selected rendition format.
.<rendition>.<renditionformat>	View mode only, specifies the rendition format, for example, pdf
.<rendition>.<app>	Specifies the full path to the default application executable including switches to use for viewing or editing the object
.<rendition>.<action>	(Optional) If the rendition invokes an action instead of an application, an <action> element is used instead of an <app> element to specify the name of the action to be invoked.
.<rendition>.<inline>	(Optional) Specifies whether the rendition can be displayed inline (view mode only). Document will be launched with HTTP content transfer mode.
.<rendition>.<label>	(Optional) The application label displayed in the format preferences UI dropdownlist. If omitted, the contents of the <app> tag will be used for the app label.
.<rendition>.<isdefault>	(Optional) Set to true to enable a rendition to be selected by default for a given mode, type, and primary format combination. Default = false.

Object types can have their own rendition settings for view and edit mode. A different application can be specified for each mode. If no default application is specified for a requested object type and format, the OS default application is used.

<modified_vdm_nodes> element

(In /webcomponent/app.xml) Sets the user's session timeout value during actions that include unsaved virtual document changes. The timeout value for the user's session will be set back to the application timeout value after the action completes.

Note: Setting the timeout value to a large number could improve performance but can also result in data loss for users whose sessions time out during a lengthy action.

Table 2-22. Modified VDM action timeout (<modified_vdm_nodes>)

Element	Description
<modified_vdm_nodes>	Contains <unsaved_changes_session_timeout>
<unsaved_changes_session_timeout>	Resets the user's session timeout in seconds when an action on unsaved virtual document nodes has begun. The default value of -1 ensures that the session does not time out until the action has completed. This may have a performance impact.

<custom_attribute_data_handlers> element

(In /webcomponent/app.xml) Contains one or more <custom_attribute_data_handler> elements that specify classes to handle custom attributes in datagrids.

<discussion> element

Contains the following elements that configure discussions.

Table 2-23. Collaborative Edition elements (<discussion>)

Element	Description
<sharing>	Sets behavior for discussion sharing between document versions. Valid values: all = discussion shared among all document versions minor = discussions shared only on minor versions none = each version has its own discussion

<xforms> element

Contains pointer to an XForms adapter service class.

Table 2-24. Business Process Manager Forms Builder elements (<xforms>)

Element	Description
<adaptorService>	Fully qualified class name

<listeners> element

Registers application, session, and request listeners. For more information, refer to [Application, session, and request listeners, page 550](#).

Table 2-25. Listener elements (<listeners>)

Element	Description
<application-listeners>	Contains one or more <listener> elements that specify a class to be notified on application startup and stop
<session-listeners>	Contains one or more <listener> elements that specify a class to be notified when each session is created and destroyed
<request-listeners>	Contains one or more <listener> elements that specify a class to be notified at each request start and end
.<class>	Fully qualified class name of the listener

<client-sessionstate> element

Contains filters that enable or disable specific client environments

Table 2-26. Client session state elements (<client-sessionstate>)

Element	Description
<client-sessionstate>	Contains one or more filters that enable or disable a client environment

Element	Description
<filter>	The value of the clientenv attribute specifies the client environment being enabled or disabled. Must match a client environment specified in <environment>.<clientenv>.
<enabled>	Set to true to enable the client environment.

<dragdrop> element

Contains elements that turn on or off drag and drop support in the Internet Explorer browser. For more information on drag and drop, refer to [Supporting drag and drop](#), page 450.

Table 2-27. Drag and drop elements (<dragdrop>)

Element	Description
<dragdrop>	Contains <enabled> element
<enabled>	Set to true to enable drag and drop in the application for the Internet Explorer browser. If set to false, the Active X plugin in the <plugins> element can still be enabled for rich text spellchecker.

<copy_operation> elements

Contain elements that determine whether to retain storage areas during copy operations. Some applications override the default setting for this feature.

Table 2-28. Copy operation elements

<retainstorageareas>	Set to true to retain storage area for objects being copied. Required for some Webtop client applications.
----------------------	--

<move_operation> elements

Contain elements that determine whether to move all versions or only the selected version during move operations.

Table 2-29. Move operation elements

<all_versions>	Set to true to move all versions of the object to the paste or drop location
----------------	--

<richtexteditor> element

Contains elements that configure the rich text editor.

Table 2-30. Rich text editor elements (<richtexteditor>)

Element	Description
<spell_checker_enabled>	Set to false to disable the spell checker in the Active-X plugin. To enable the spell checker, set to true and set the value of <enhanced_plugin>.<enabled> to true. The spell checker requires Microsoft Word on the client.

<plugins> element

Contains elements that enable the spell-check dictionary and drag and drop Active-X plugin for Internet Explorer.

Table 2-31. Active-X plugins elements (<plugins>)

Element	Description
<enhanced_plugin>	Contains <enabled>, <classid> and <min_version> elements

Element	Description
<enabled>	Set to true to enable the rich text dictionary and drag and drop plugins for Internet Explorer in the application. If disabled and <dragdrop> is enabled, drag and drop within WDK applications is supported but not to and from the Desktop. For desktop drag and drop, both this element and <dragdrop> must be set to enabled.
<classid>	Specifies the Active-X plugin classid
<min_version>	Specifies the plugin major and minor version
<initial_user_state>	Sets the initial plugin state for the user before a preference is set. With the default value of false, the user must set a preference to enable the plugin download. With a setting of true, all users must have privileges that allow them to install Active-X plugins. If the plugin is deployed by SMS, the initial user state should be set to true.

<display> element

The <display> element configures the display of hidden objects for the application. Set <display>.<hiddenobject> to true to display hidden objects.

<applet-tag> element

The <applet-tag> element configures the rendering of applets. The following elements configure applet rendering in the application:

Table 2-32. Applet tag elements (<applet-tag>)

Element	Description
<mode>	Applet is rendered as either HTML applet (deprecated in HTML 4.01) or HTML object. The object tag allows control of the version of the Java plugin used by IE on Windows clients. Valid values: applet object
<plugin-manual-install>	Specifies a complete URL for download of the browser plugin for browsers that do not have a Java plugin installed, for example, http://www.java.com . The URL will be displayed in browsers that do not have the plugin.
<activex-classid>	For object applets, Windows/IE only. Identifies the Active-X class ID of the specific version of the Java plugin to be used, for example, CAFEEFAC-0014-0002-0008-ABCDEFEDCBA corresponds to Java plugin version 1.4.2_08. If no value is present, the latest plugin is used. (The Java plugin for IE is written as an Active-X control.)
<activex-install>	Specifies a URL for automatic installation of the Java plugin, for example, http://java.sun.com/products/plugin/autodl/jinstall-1_4_2-windows-i586.cab#version=1,4,2,8 . Version is the minimum supported by the application. If the user has a lower version, download will be triggered. If the URL is prefixed with "/", the path is relative to the application root directory, and user must have appropriate read permissions on that location.

<job-execution> element

This element and its contained elements can be added to an application configuration file to support global settings for asynchronous jobs (actions and components).

Table 2-33. Asynchronous job elements (<job-execution>)

Element	Description
<job-eventhandler>	Fully qualified class name of default event handler for actions and components
<async>	Contains settings that override the individual action or component asynchronous setting
<enable-async-job>	A value of false turns off asynchronous processing for the application. By default each individual component and action job will process synchronously unless <enable-async-job> has a value of true and the component or action definition sets <asynchronous> to a value of true.
<sendnoticeonfinish>	Set to true to send a notice to the user's inbox when a job has finished
<async-jobs-max-limit>	Integer value that limits the number of asynchronous jobs a user can run concurrently. A value of 0 indicates no limit.

For more information on asynchronous jobs, refer to [Asynchronous action and component execution](#), page 571.

Web deployment descriptor (web.xml)

A deployment descriptor file is defined and described by the J2EE Servlet specification. WDK-based applications provide a customized web.xml file that specifies the top application layer and the mapping of servlets that are used by WDK. Additionally, the WDK installer and installers for WDK client applications such as Webtop modify the root web application web.xml file if virtual link support is requested during installation (refer to [Virtual links](#), page 91).

The first set of elements in a web deployment descriptor are context parameters for the application. The WDK context parameters are described in the features to which they apply:

Table 2-34. Context parameters

Context Parameter	Description
AppFolderName	Specifies the top layer for the application, for example, custom. For more information, refer to Application name, page 58 .
StaticPageIncludes	Specifies the file extensions that do not need to be processed by the WDK controller filter.
StaticPageExcludes	Specifies paths to files whose extensions are listed in StaticPageIncludes but which should be treated as dynamic (refreshed), that is, exceptions for the general file extension. Uses regular expression syntax. For example, the path <code>/*/formaticon/*/fileExt/*/file.gif</code> , which excludes all format icons in each theme, is written as follows: <![CDATA[^.*?\/formaticon\/.+?\/fileExt\/.*?\/file\.gif\$]]>
HTTPSessionRequired	Specifies URLs that do not require a new HTTP session. Uses regular expression syntax. The default value specifies that URLs to UCF will not create a new HTTP session, because UCF on the client could issue heartbeats that do not require a session. For information about Apache expression syntax, refer to the Apache Web site .
UseVirtualLinkErrorPage	Set to true to use the VirtualLinkHandler servlet for all HTTP document requests that result in a HTTP "404 - File Not Found" error. This servlet tries to resolve the request as a virtual link.

To add a static page type that will not be processed:

1. Open the web deployment descriptor (/WEB-INF/web.xml).
2. Change the value of the StaticPageIncludes context parameter to include the static page extension. The following example adds the sound file extension .wav to the list:

```
<context-param>
```

```

<param-name>StaticPageExtensions</param-name>
<param-value>
  <![CDATA[(\ .js|\ .css|\ .gif|\ .jpeg|\ .jpg|\ .html|\ .htm|\ .bmp|\ .wav)$]]>
</param-value>
</context-param>

```

The following table describes the servlet filters that are defined in web.xml:

Table 2-35. WDK filters

Filter	Description
WDKController	Maps all requests ("/") but does not process requests for static pages as specified in the context parameter StaticPageExtensions. Initializes the config service, binds session, request, and response objects to current thread, sends notifications to application, session, and request listeners. Detects failover and notifies components upon recovery. For more information about failover, refer to Implementing failover support, page 442 .
RequestAdapter	Processes requests and intercepts requests with multipart/form-data encoding in the header.
UcfSessionInit	Binds the UCF manager instance to the HTTP request/response context
CompressionFilter	Compresses text responses for configured file extensions. For more information, refer to High latency and low bandwidth connections, page 350 .
ClientCacheControl	Limits the number of requests by telling the client browser to cache static elements. For more information, refer to High latency and low bandwidth connections, page 350 .

The following listener is specified in web.xml:

Table 2-36. Deployment descriptor listener

Class	Description
NotificationManager	Instantiates all classes that implement IApplicationListener, such as the EnvironmentService class. Used by the failover mechanism.

The following table describes the servlets that are defined in web.xml:

Table 2-37. WDK servlets

Servlet	Description
UcfGAIRConnector	Specifies the servlet that uses the GAIR protocol to communicate data between the client and the application server
UcfInitGAIRConnector	Specifies the servlet that initializes the GAIR connector
UcfNotification	Specifies a servlet that implements INotificationHandler, which UCF uses to notify the client of errors and progress
WorkflowEditorServlet	Displays a Web-based workflow editor
VirtualJS (in root application web.xml file)	Rewrites the static WDK JavaScript files to portlet-specific versions, changing method names and any form variable names that are used to the appropriate Portal server namespace. .
Trace	Servlet for tracing WDK
ComponentDispatcher	The component dispatcher maps a URL to a component to the appropriate component start page.
ActionDispatcher	The action dispatcher maps a URL to an action to launch the action.
DRLDispatcher	Converts a DRL to a URL
SessionTimeoutControl	Overrides the JSP container timeout to provide finer timeout management
wdk5-contentsender	WDK 5 content transfer servlet
wdk5-contentreceiver	WDK 5 content transfer servlet
wdk5-download	WDK 5 content transfer servlet that streams browser-supported content to the browser.

Servlet	Description
wdk5-xmlutil	WDK 5 content transfer servlet that detects links in XML files.
wdk5-appletresultsink	WDK 5 servlet used by content transfer applets to return results
HttpContentSender	Wraps HttpSessionServlet
FileFormatIconResolver	Returns docbase format icon for given file extension
DownloadServlet	Used by HttpContentTransportManager to stream content to the browser
VirtualLinkHandler	Class that handles File Not Found (404) errors, passing them to the virtual link servlet. This allows a different servlet for each WDK application in the server instance.

The following `errorpage` element is defined in `web.xml`. Two error pages are specified: one to handle 404 (Page not found) errors by the virtual link handler servlet, and one to handle 500 (Internal Server Error) errors. The error JSP page for 500 errors sets the response status to 200 to prevent the application server from displaying the stack trace and revealing application internals.

Table 2-38. `<errorpage>`

<code><error-code></code>	Specifies the error codes that will be handled by the specified error page
<code><location></code>	Specifies the servlet that handles the error code

The following elements are added to the Web deployment description (`web.xml` file) for the root Web application on the J2EE server if virtual link support is requested during installation.

```

<web-app>
1<context-param>
    ...
    2<servlet>
        <servlet-name>VirtualLinkHandler</servlet-name>
        <servlet-class>com.documentum.web.virtuallink.VirtualLink404Handler
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>VirtualLinkHandler</servlet-name>
        <url-pattern>/VirtualLinkHandler</url-pattern>
    </servlet-mapping>
    3<error-page>
        <error-code>404</error-code>

```

```

        <location>/VirtualLinkHandler</location>
    </error-page>
</web-app>

```

- 2 Defines the virtual link servlet name, class, and URL pattern mapping
- 3 Defines the HTTP error code and handler. (The virtual link handler intercepts all 404 File Not Found errors.)

Application environment properties

The `Environment.properties` file, located in `/WEB-INF/classes/com/documentum/web/formext`, sets some application listener classes and other application-wide settings. The following table describes the settings in the environment properties file.

Table 2-39. Environment settings

Setting	Description
LookupHookPath.# LookupHookArgument.# LookupHookClass.#	Sets a path, scope argument, and class name for each configuration service listener class. Refer to Configuration lookup hooks, page 482 .
SessionHookClass	Session listener class that implements <code>ISessionHook</code> in <code>com.documentum.web.formext.session</code>
ComponentServletPath	Used by <code>Component.getServletPath</code> to resolve the path statically rather than dynamically. Should match the <code>ComponentDispatcher</code> servlet mapping in <code>web.xml</code> .
ConfigReaderClass	Class that parses XML configuration files
non_docbase_component.#	Name of component that does not require a Documentum session (login is not presented)
DocbaseFolderTreeShowMoreThreshold	Limits the number of folders that will be displayed in the tree. A larger number of folders will result in a More Folders link instead of a listing of all folders. Set this number to optimize performance of the tree.

Configuring application failover support

Session failover is required in a clustered application server environment. User session data is persisted, and the load balancer routes the last HTTP request before failover to the secondary server.

The WDK infrastructure detects failover and provides recovery by notifying components of failover. Components can then perform cleanup and recovery. Failover is configurable and can be implemented by components. Data integrity is preserved during failover for components that implement failover.

The following components support failover in this release:

- Container components wizard, combo, property sheet, and property sheet wizard (not content transfer containers) will persist completed form data for the user's session.
- Advanced search criteria and location are persisted
- The state of property editing components is persisted

Note: The following features do not support failover: inbox, navigational history, content transfer components, asynchronous job state, and changes to virtual document structure.

The following topics describe failover configuration:

- [Configuring application-wide failover, page 89](#)
- [Configuring component failover, page 90](#)

Refer to [Implementing failover support, page 442](#) for information on implementing failover in custom components.

Configuring application-wide failover

Application failover is enabled in a setting of the WDK app.xml file. The setting `<failover>.<enabled>` turns on serialization for all failover-enabled components and sessions in the application. This setting is filtered for the defined client environments, so that failover can be disabled for environments such as portals that do not support failover. If you migrate WDK 5.2.5 customizations to WDK 5.3, and your custom classes do not support failover, you can disable failover in your custom app.xml file, or you can run a mixed environment, with some components supporting failover and some not, just as in the current Webtop.

Tip: Some application servers have a configuration setting that enables or disables session serialization. In Tomcat version 5.x, serialization is turned on by default and you will see error messages for non-serializable objects. This feature of Tomcat makes it useful for identifying objects that should be marked transient in your preparations

for failover support. To turn off the default serialization, refer to the Tomcat Server Configuration Reference documentation.

To disable failover support in the application:

1. Open the application definition file (/custom/app.xml).
2. Add the failover element, filter for the appropriate client environment, and override the value. For example:

```
<failover>
  <filter clientenv="not portal">
    <enabled>true</enabled>
  </filter>
</failover>
```

The failover filter is described in [Implementing failover support, page 442](#).

Configuring component failover

The Control class, and its subclasses such as Component, implements the Serialized interface. Any component can potentially support failover.

Each component that supports failover must have a configuration element <failoverenabled> with a value of true:

```
<failoverenabled>true</failoverenabled>
```

All components in a container must support failover in order for the container to support failover.

If a component is marked as failover-enabled and is extended by another component, the extended component will inherit failover support. If the extended component needs to do additional work for recovery or cleanup, it must override onRecover(), call super.onRecover(), and do the additional work.

Containers that are failover-enabled override onRecover() and call onRecover on all contained components, which in turn call onRecover() on their controls.

Note: If a container is marked as failover-enabled and contains a component that is not failover-enabled, WDK will not serialize the container.

If the user is on a component that does not support failover, and failover occurs, the application home page will be displayed after recovery.

Configuring content transfer mode for the application

Content transfer in WDK 5.3 supports three modes:

- Custom components that extend WDK 5.2.5 content transfer components (applet-based)
- HTTP content transfer
- Unified Client Facilities (UCF), a lightweight client-based service

For information on the features available with each mode, refer to [Content transfer modes compared](#), page 510.

To configure your application to use either UCF or WDK 5.2.5 applet-based content transfer, set the value of `<contentxfer.>.<default-mechanism>` in `app.xml` to `ucf`. To use HTTP-based content transfer, set the value to `http`.

If the application is configured to use UCF content transfer, the lightweight UCF applet downloads to the client on the first call to a content transfer operation, unless the application has been configured to support HTTP transport only. The applet is lightweight so that it can be downloaded every time it is needed and does not need to be installed. This removes the security restriction for users that do not have permission to install applets.

Clients must have the Sun Java plugin installed in the browser to support UCF. HTTP content transfer requires either the Sun or Microsoft browser JVM. The JVM is not installed by default in some versions of IE, so you must install the JVM if one is not present. The JVM must be of the supported version that is specified in the release notes for your WDK product.

For more information about UCF, refer to [Unified client facilities \(UCF\)](#), page 513.

WDK 5.2.5 content transfer components are present in the WDK runtime for backward compatibility. They cannot be addressed directly by URL because they are versioned, but if your custom component extends a WDK 5.2.5 component definition, it will work.

Virtual links

The virtual link service supports virtual links, which are URLs to view a single document with the following syntax:

```
http(s)://server:port[/repository:/virtual_directory/
folder_path]/object_name?format=dmformat_name
```

For example, the virtual link service can resolve the following kind of link:

```
http://localhost:8080/webtop/mydocbase:/somecabinet/somedoc
```

or

```
http://localhost:8080/webtop/somecabinet/somedoc
```

The virtual link service processes the virtual link by providing authentication and resolving the URL path to a document in the repository.

Virtual link support is installed with every WDK application.

Virtual links are resolved to a matching object in the named repository for the named application. To handle URLs without the the application virtual directory, you must install virtual link support to a default WDK application, for example:

```
http://localhost:8080/somecabinet/somedoc
```

This option in the installer is called the **Root virtual link handler**. The virtual link handler is installed in the application server root or default Web application (in addition to the WDK application). Only one WDK application on the application server can then handle URLs that do not specify the application virtual directory.

RightSite URLs are also supported by the **Root virtual link handler** option. The following kinds of RightSite URLs will redirect to the default WDK application, that is, the WDK application that has legacy RightSite support installed:

```
http(s)://server:port/repository-name:/folder-path/  
    ../objectname  
http(s)://server:port/RightSite/repository-name:/  
    folder-path/.../objectname  
http(s)://server:port/rs-bin/RightSite.dll//  
    folder-path/.../objectname
```

Note: If an object name contains non-ASCII characters, the virtual link will not be resolved, as URLs are limited to ASCII characters.

The virtual link service consists of a `virtuallinkconnect` component and a virtual link handler servlet, `VirtualLinkHandler`. The service will handle any failed HTTP document request that results in an HTTP 404 File Not Found error by attempting to resolve the failed request to a document in a repository.

The virtual link service is described in the following topics:

- [Virtual link handler deployment, page 92](#)
- [Virtual link connection and authentication, page 93](#)
- [Virtual link path resolution and document delivery, page 94](#)
- [Virtual link error handling, page 96](#)

Refer to *Web Development Kit Reference Guide* for information on configuring the `virtuallinkconnect` component.

Virtual link handler deployment

When a WDK-based application is installed, the installer installs virtual link support for the current application. The installer also provides an option to install the root virtual

link handler in the root or default Web application provided by the application server. If you install the virtual link application in the root Web application, virtual link requests that are not prefixed with an application name, and Right-site style URLs, are handled.

Note: The virtual link handler servlet must be installed on the same host as the Web application.

If you install the root virtual link handler, you must specify the name of the default WDK application to be used for user authentication. The default WDK application will be used to attempt authentication for a virtual link when the user does not have a current session or the URL does not specify an application virtual directory. In the following example, the user has installed Webtop with the virtual directory alias wt53 and root virtual link support. The installer writes the name of the default WDK application to the root web application web.xml file as follows:

```
<context-param>
  <param-name>DefaultWdkAppName</param-name>
  <param-value>wt53</param-value>
</context-param>
```

Note: You can modify the default WDK application.

The virtual link handler servlet is registered in the web.xml file of the WDK-based application, and optionally in the root Web application, as follows:

```
<servlet>
  <servlet-name>VirtualLinkHandler</servlet-name>
  <servlet-class>com.documentum.web.virtuallink.VirtualLink404Handler
</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>VirtualLinkHandler</servlet-name>
  <url-pattern>/VirtualLinkHandler</url-pattern>
</servlet-mapping>
```

Virtual link connection and authentication

When a user supplies a virtual link URL in the browser, the virtual link handler checks for authentication in the following order:

- Credentials passed with the request
- Current WDK session
- Virtual link anonymous account
- Login dialog

If this is the first request for a virtual link during the user's session, the handler determines the WDK application to redirect to by looking for an application cookie in browser memory. If the handler does not find a cookie or an application virtual directory in the URL, it reads the default application name from the root web application

web.xml file. The handler then redirects to the `virtuallinkconnect` component in the WDK application.

You can configure anonymous access for virtual links through the `virtuallinkconnect` component definition. Only one anonymous account per repository is supported. Each application in the application server instance must use the same anonymous account for a given repository. For example, if the DA virtual link uses anonymous account A for repository A, the Web Publisher virtual link must use account A for repository A. The repository must have a guest account for which the username and password match those in the `virtuallinkconnect` component definition.

Setting up an anonymous virtual link account

1. Encrypt the anonymous virtual link account password for the repository with the trusted authenticator tool (refer to [To use the password encryption tool](#); page 105).
2. Copy `/wdk/config/virtuallinkconnect_component.xml` to `/custom/config` and open the file for editing.
3. Paste the encrypted password into the `<defaultaccounts>` element. (Add one `<defaultaccount>` element for each repository.) For example:

```
<defaultaccount>
  <filter docbase='repository_name'>
    <docbase>my_repository</docbase>
    <username>default_user</username>
    <password>d7d1d6e383d6d4e1d0</password>
    <domain></domain>
  </filter>
</defaultaccount>
```

4. Set up the guest account in the repository using Documentum Administrator.

The `virtuallinkconnect` component reads the list of root paths for the authenticated repository from the component definition and constructs a virtual link URL using the original request, adding rootpath information, authentication arguments and an optional format argument that retrieves a rendition.

Refer to *Web Development Kit Reference Guide* for information on configuring the `virtuallinkconnect` component.

Virtual link path resolution and document delivery

A virtual link has the following syntax:

```
http:// host_name/virtual_directory[/repository:/path]/
document_name?format=dmformat_name
```

For example, the following virtual link resolves to an HTML rendition of a document in `my_repository`:

```
http://myserver/virtual_directory/my_repository:/mycabinet/mydoc?format=html
```

The repository portion of the link is optional. If it is not supplied, the current repository is assumed. If the virtual directory is not supplied, then a root virtual link handler must resolve the application name.

Preferred rendition and document format — If the user does not provide a format or extension, the virtual link handler will attempt to get the user's preferred rendition for the document. If the user supplies an extension for the file name in the virtual link, the link handler will attempt to locate a rendition in the requested format. For example, the following virtual link requests a document in PDF format:

```
http://webtop/MyCabinet/MyFolder/MyDocument.pdf
```

The link handler will look for a PDF rendition of the document. If there is no match, the handler will remove the extension and look for the object. If no object with the corresponding path and name is found, the handler returns an error message.

A virtual link will resolve to the current version of the document in the same location. If a more recent version has been created in another location, the virtual link will resolve to the most recent version in the location specified by the link. For example, a virtual link is created that points to version 1.1 of a document in folder A. Then a user creates a version 1.2 of the document and moves that version to folder B. The original virtual link that points to version 1.1 in folder A will return the version 1.1 document even though it is not the current or latest version. If, however, version 1.2 is in folder A, the same folder as the original link, version 1.2 will be returned by the link.

Document path — The document portion of the link corresponds to the document name. The path may not be the full repository path to the document. The virtual link handler will attempt to use the rootpaths that are defined in the `virtuallinkconnect` component to resolve the full path to the document.

After the virtual link handler has authenticated the user, the handler matches the document path in the virtual link URL by retrieving the list of rootpaths from the `virtuallinkconnect` component definition. The full URL is formed by concatenating the first rootpath with the path on the virtual link. If the document is not found in that folder, the next rootpath is tried. If no match is found with any rootpath, the handler tries the virtual link as an absolute repository path, for example, `http://myhost/Cabinet1/folderA/DocumentX.html`.

Note: A virtual link URL should not be manually formed with the arguments specified. The only exception to this rule is the `format` argument, which should be specified manually if a specific content format is required.

If no document is found with this algorithm, or the user does not have permission to read a document, an HTTP 404 File Not Found error is returned to the browser.

Inline document links — If a document is presented for viewing, and the document contains links (such as a Microsoft Office or Adobe PDF document), the links will work

only if they are links that were created based on the folder structure of the document inside the repository. For example, if the user is viewing a document MySecrets in /My Cabinet/My Folder 1/My Sub Folder 1, and this document contains a link to another document in the same folder, the linked document will be displayed when the user clicks the link. When the user clicks a link like "My Sub Sub Folder 1/my other document.pdf", the file "my other document.pdf" located at /My Cabinet/My Folder 1/My Sub Folder 1/My Sub Sub Folder1 will be displayed.

If the requested document is a virtual document or complex document, only the parent document will be returned.

Virtual link URLs and content transfer — Virtual links must have ASCII characters in the URL path, in accordance with the URI syntax as defined in World-wide Web Consortium RFC 2396. Repository folder and cabinet names that contain non-ASCII characters cannot be resolved by a virtual link.

When a document is matched, the virtual link handler checks for a format URL argument to specify the rendition that is requested. The handler then writes the content to the browser, bypassing the WDK content transfer mechanism, and sets the HTTP content header to the mime type that correspond to the format of the content.

Virtual link error handling

The web.xml deployment descriptor is modified to redirect all HTTP 404 (not found) errors to the virtual link handler as follows:

```
<error-page>
  <error-code>404</error-code>
  <location>/VirtualLinkHandler</location>
</error-page>
```

In addition to handling 404 (Page Not Found) errors, the virtual link error page will report errors for invalid user credentials, invalid formats, or generic errors. This virtual link error page is used when the UseVirtualLinkErrorPage context parameter is set to true in the application /WEB-INF/web.xml file:

```
<context-param>
  <param-name>UseVirtualLinkErrorPage</param-name>
  <param-value>true</param-value>
</context-param>
```

The virtual link error page is constructed by the virtual link handler using messages that are specified in /wdk/strings/com/documentum/web/virtuallink/VirtualLinkNlsProp. properties.

Content server event notification

Users can request notification for a Content Server event on one or more subscribed objects in the subscriptions UI. All users who are subscribed to the object and have turned on notification for the event will receive a notification in the Documentum inbox, and an email to Microsoft Outlook if a WDK application was installed with Outlook integration.

To configure the list of Content Server events that are available to the user for notification, add each event to the <notification> element in your custom app.xml file. (For information on the app.xml configuration elements, refer to [<notification> element, page 72.](#)) Any API, workflow, or lifecycle event can be added to the configuration. Notification is available on dm_sysobject and its subtypes. If an event that is registered in app.xml does not exist in a particular repository, that event will be ignored by the event notification mechanism for users logged into that repository.

For information on the existing Content Server events, refer to *Content Server API Reference Manual*.

Note: Event notification on replica or reference objects is not supported and will not be permitted in the subscription UI.

Navigation defaults

You can set properties of the form processor that affect memory usage and URLs that are returned for specific cases. The form processor properties are defined in /WEB-INF/classes/com/documentum/web/form, in the file FormProcessorProp.properties. The configurable properties are described in the following table.

Table 2-40. Navigation settings

Property	Description
manageBrowserHistory	True to maintain browser history (pages that the user has navigated from)
processorHookClass	Specifies the fully qualified class name of a form processor hook class
timeoutURL	Specifies a URL to a page to be displayed when the user times out
historyReleasedURL	Specifies a URL to a page to be displayed when the user attempts to navigate back beyond the browser history

Property	Description
noReturnURL	Specifies a URL to a page to be displayed when the user attempts to return to a page or component that has no caller, for example, the first URL used to connect to the application
serverBusyURL	Specifies a URL to a page to be displayed when the number of HTTP sessions has been exceeded (<max_sessions> value in app.xml)
requestHistorySize	Specifies the number of URLs maintained as browser history when manageBrowserHistory equals true
formProcessorClass	Class that performs the form processor functions
eventHandlerSessionTimeout	Number of minutes to keep the HTTP session alive during event handling. Overrides the session timeout specified in web.xml. For example, if a delete operation for many objects is expected to take up to 4 hours to complete, increase this value to 240.

Browser history

When the user navigates away from a component JSP page using the **Back** or **Forward** button, the user's selections on the page are lost. If the state should be saved, set the keepfresh attribute on the <dmf:form> tag to true in the JSP page.

You can turn off browser history for Web applications that do not need to maintain navigation history. You can configure the number of pages that are retained in history. This value is configurable so that you can tune memory usage, since memory is consumed by maintaining history for each browser window in each user session.



Caution: If you turn off browser history, some controls may not work properly when the user navigates using the browser **Back** button.

To configure browser history

The form processor has configurable properties in the file FormProcessorProp.properties, which is located in /WEB-INF/classes/com/documentum/web/form. You can turn on

browser history management and set the number of page requests that are retained for browser history.

1. Open `FormProcessorProp.properties` in `Application_Root/WEB-INF/classes/com/documentum/web/form`.
2. Set the value of `manageBrowserHistory` to `false` to serve each page request in a single network round trip. The browser will not necessarily have the correct URL for the page it is currently displaying. Set the value to `true` to manage browser history through the history mechanism. This closes browser history around nested forms so that the user cannot return to a nested form after leaving the nest.
3. Set the value of `requestHistorySize` to specify the number of snapshots that will be held in history for each window or frame (default = 10). If the value is empty or zero, there is no limit to the number of snapshots that can be kept in the collection, which can significantly affect performance.

Tip: Name all user input controls and controls that must maintain state when the user navigates back to them. Only controls that are named are saved in a snapshot and retrieved in browser history.

Cookies

Cookies are used by WDK-based applications to store login information, user preferences, and other application data on the client machine. Cookies are encoded so that they are not stored as plain text. Some session cookies are in-memory only and are not stored on the client.

Refer to [Storing and retrieving user preferences, page 567](#) for details on reading and writing your own cookies.

Cookie lookup can slow performance. To improve performance, cookie preferences are stored in memory the first time they are read or written in a session.

Login, SSOLogin, and GeneralPreferences write several cookies to the client browser that set user preferences:

Table 2-41. Preferences cookies

Cookie	Description
LOCALE_CONFIG_PATH	User's selected locale
CONFIG_USERNAME	Name of user who is logged in

Cookie	Description
CONFIG_SHOWOPTIONS	Sets whether to display the extra login options: domain, locale, and accessibility settings
DOMAIN_CONFIG_PATH	Name of selected repository domain
DOCBASE_CONFIG_PATH	Name of selected repository
CONFIG_ENTRYPAGE	Sets user's preferred Webtop view (classic or streamline)
CONFIG_ALTTEXT_ENABLED	Sets user's preference for alt text (accessibility mode)
CONFIG_KEYBOARDNAVIGATION_ENABLED	Sets user's preference for keyboard shortcuts (accessibility mode)
CONFIG_SHORTCUTNAVIGATION_ENABLED	Sets user's preference for shortcuts to the top and bottom of the navigation tree (accessibility mode)
theme	Sets user's preferred display theme
INHIBIT_CHANGE_LOSS_WARNING	ComboContainer preference that inhibits the warning when the user cancels checkout
INHIBIT_CONFIRM_PROMPT	Sets user preference for the display of a confirmation prompt upon delete
PREF_SHOW_THUMBNAILS	Set's user's preference to display thumbnail images
m_strPreference	Sets user's selected data page size

A set of cookies are stored for internal purposes:

Table 2-42. Internal cookies

Cookie	Description
JSESSIONID	ID for the user's session, generated by HttpServletRequest
FULL_INSTALL_ARCHIVE_COOKIE_NAME	Sets a boolean cookie that signifies that the full content transfer applets have been installed
INSTALL_SUCCESS_COOKIE_NAME	Sets a cookie that signifies that the content transfer applets installed successfully and do not need to be downloaded again

Cookie	Description
CODEBASE_URL	File selector applet cookie that specifies the URL to the applet codebase
PARAM_BROWSER_FILTER	File selector applet, last filter used
PARAM_BROWSE_DIR	File selector applet, user's default browse directory
PARAM_RESULT_KEY	File selector applet, result key to retrieve results
PARAM_RESULT_SERVLET_URL	File selector applet, URL of the result servlet
PARAM_SESSION_ID	File selector applet, HTTP session ID
PARAM_STYLE_INFO_URL	File selector applet, URL to applet style
PARAM_VALUE	File selector applet, applet value
PREFERENCE_LAST_DIRECTORY	Stores the last directory selected by user during import with the file selector applet
CONFIG_ADD_FOLDERS_ENABLED	Boolean setting to enable Add Folders in file selector applet
appname	Name of Web application. For example, the name of the application at http://localhost/webtop is webtop.
isMicrosoftVm	IE cookie that is read by Browser to determine the user's client VM

Timeout

The timeout of your Web application is managed through the J2EE server. The J2EE servlet specification supports a `<session-timeout>` element in the `web.xml` deployment descriptor file. Locate the `<session-config>` element in `/WEB-INF/web.xml` and change the timeout value (in minutes). For example:

```
<session-config>
  <session-timeout>60</session-timeout>
</session-config>
```

Repository timeout settings are configured through the `dmcl.ini` file on the J2EE server host. Login ticket expiration settings are in the server config object. Refer to *Content Server Administration Guide* for details on client and login ticket timeout settings.

A timeout page, `timeout.jsp`, is displayed for HTTP session timeout. Modify this page to redirect the user to a login page or other timeout component. In the following example, the virtual root global variable is resolved by the `<dmf:webform>` tag in the JSP page:

```
function loginRedirect()
{
    getTopLevelWnd().location.replace( g_virtualRoot+"/component/main");
}
```

Operations timeout — The form processor has a property that overrides the HTTP session timeout. The `eventHandlerSessionTimeout` property is used to set timeout in minutes during event processing. For example, if a delete operation for many objects is expected to take up to 4 hours to complete, increase this value to 240. This property is found in `/WEB-INF/classes/com/documentum/web/form/FormProcessorProp.properties`.

Browser unload timeout — You can override the user's HTTP session timeout when the client browser has closed without an explicit logout. When the user closes the browser window or navigates to an outside URL, the top frame unload event is triggered. A timeout servlet is called when the main frame of the application is unloaded.

This timeout control can be traced with the flag `SESSIONTIMEOUTCONTROL`.

To override the session timeout, perform the following steps:

1. Open the JSP page that contains your application top-level frameset.
2. Include the timeout control JavaScript file as follows:

```
<script language="JavaScript1.2" src='<%=Form.makeUrl(
    request, "/wdk/include/timeoutControl.js")%>'>
</script>
```

3. Include a hidden frame in the top level frameset. In the following example, the top frame calls the `manageTimeout()` JavaScript function in the JavaScript file `timeoutcontrol.js` when the top frame is unloaded:

```
<frameset rows="0,38,*,30" border="0" framespacing="no" frameborder="no"
    onunload='manageTimeout(frames["timeoutcontrol"])'>
    <frame name="timeoutcontrol" src='<%=Form.makeUrl(
        request, "/wdk/timeoutcontrol.jsp?Reload="
        + System.currentTimeMillis()%>' marginwidth="0" marginheight="0"
        frameborder="no" border="0" scrolling="no" noresize>
    </frame> ...
</frameset>
```

4. Configure the timeout override parameters in `app.xml`. Add an element `<session_config>.<timeout_control>` as a child element of `<application>`. The value of `<client_shutdown_session_timeout>` specifies the number of seconds before the session will be shut down after the main frame has been unloaded by user action. For example:

```
<session_timeout_control>
```

```

    <client_shutdown_session_timeout>60
  </client_shutdown_session_timeout>
</session_timeout_control>

```

The default value is 120 seconds if no configuration element is present, and the minimum is 15 seconds. If the timeout is larger than the actual HTTP session timeout configured in web.xml, the session timeout will not be overridden.

Virtual document operations timeout — The setting `<modified_vdm_nodes>`. `<unsaved_changes_session_timeout>` in `/webcomponent/app.xml` overrides the session timeout when a user has launched an operation on unsaved virtual documents. The default value is `-1`, so that the session does not time out during the operation. After the operation has completed, the timeout is reset to the value in web.xml.

You can copy the parent element `<modified_vdm_nodes>` and its child to your custom `app.xml` and override this setting.

Note: Setting the timeout value to a large number could improve performance but can also result in data loss for users whose sessions time out during a lengthy action.

Application login and authentication

Several types of login connections are supported in the WDK authentication framework:

- [Per-session authentication \(login dialog\), page 104](#)
- [J2EE principal authentication, page 104](#)
- [Single sign-on, page 107](#)
- [Ticketed login, page 108](#)

Refer to the following additional login topics:

- [Skip authentication, page 110](#)
- [Silent login, page 542](#)
- [Explicit login, page 111](#)
- [Login preferences, page 111](#)
- [Login locale, page 111](#)
- [Number of user sessions, page 112](#)

The WDK framework employs lazy authentication in which authentication occurs on the first access to a specific repository. If your custom application employs a login dialog, you can force authentication by calling the `authenticate()` method in the `Login` class.

If a component is called by URL or Java method, the component dispatcher determines whether the user has a valid Documentum session. If there is no session, the dispatcher calls the authentication service, which attempts authentication using authentication schemes in the order specified in the

com.documentum.web.formext.session.AuthenticationSchemes.properties file: per-session or manual login, single sign-on, ticketed login, or J2EE principal login. Place your preferred authentication scheme first in this list. If none of these authentication schemes succeeds, the dispatcher calls the login component.

In addition to the topics listed above, refer to [Table 2-7, page 64](#) for a description of the authentication configuration elements in app.xml. For information on customizing login, refer to [Authentication service, page 539](#).

Per-session authentication (login dialog)

The WDK framework employs lazy authentication in which authentication occurs on the first access to a specific repository. The user will be presented with a login dialog when they connect to a component that requires a session.

In a portal environment, a user must first authenticate against the Portal environment via the portal's login page and then authenticate against each Content Server via a WDK login page.

Per-session authentication logs a user into a repository when the user supplies the username, domain (if required), and repository. The user's entries, except for password, are stored in a cookie for subsequent login default values.

First-session authentication uses the same scheme. After a successful Content Server connect it will store the password in the portal server's preference store by making a call to the environment's `IPreference.writeString()` method.

If your custom application employs a login dialog, you can force authentication by calling the `authenticate()` method in the Login class.

J2EE principal authentication

J2EE principal-based authentication allows a single login to the Web server and the Content Server. Each J2EE-compliant server has its own documented procedures for setting up server-based authentication. WDK supports server-authenticated users by means of a trusted authenticator login to each repository. The identity of the user who logs in to the Web application must match the login identity in the repository. This identity (username) is passed to the Web application, but the user's password is not passed. WDK then logs into the repository for the user by employing a trusted authenticator identity. The trusted authenticator must be a superuser for the given repository.

To set up J2EE principal authentication

1. Make sure that J2EE principal authentication is listed first in the list of authentication schemes in the `com.documentum.web.formext.session.AuthenticationSchemes` properties file. Authentication will be attempted in the order that they are listed. For example, if repository authentication is listed first, a login dialog is always presented.
2. Encrypt the superuser's password and paste the encrypted form of the password into `web.formext.session.TrustedAuthenticatorCredentials.properties`. Refer to the steps below for encrypting the password.
3. Set up J2EE principals in the application deployment description `web.xml` and in application server-specific files. Refer to the instructions below for information on modifying `web.xml`. Refer to the server documentation for application-server specific setup.
4. Stop and restart the application server to enable J2EE authentication.

In a portal environment, user principal authentication requires that the user log on to the portal. The portal user name must match the repository user name, although the passwords do not have to match. After authentication with the portal, a WDK session is established automatically and the user can access the Content Server through the WDK portlet components. The user's privileges in the repository are assigned through the user's role or permissions, so that the user does not acquire the superuser's privileges. The default or preferred repository for the user is stored automatically the first time the user logs on. This can be changed manually using the Documentum portlet preferences.

The WDK framework uses the Content Server ticketing mechanism to obtain a ticket for a Superuser. The actual user name, and the Superuser's ticket, are used to establish a connection for the user. The user's identity remains authenticated until a new identity for the same repository is provided or the Documentum session terminates via HTTP session timeout or client logout.

To use the password encryption tool:

You can encrypt the Superuser's password for the repository with the trusted authenticator tool (`com.documentum.web.formext.session.TrustedAuthenticatorTool`). This tool uses a Caesar cipher for encryption. You can use your own tool for encryption as well.

1. From the command line, with `com.documentum.web.formext.session.TrustedAuthenticatorTool` and the Java SDK in your classpath, run the following command on a single line. Substitute the actual repository password to be encrypted:

```
java -classpath "%CLASSPATH%;T:\app\WEB-INF\classes"
  TrustedAuthenticatorTool password
```

The output will look similar to the following:

```
Encrypted: [d7d1d6e383d6d4e1d0], Decrypted: [my.pwd6\]
```

2. Paste the encrypted form of the password into `web.formext.session.TrustedAuthenticatorCredentials.properties`. Each repository must have three entries (substitute the actual repository name in the sample entries below):

```
Repository_name.user  
Repository_name.password  
Repository_name.domain
```

If no domain, then type the following:

```
Repository_name.domain=
```

For example:

```
mydocbase.user=superuser1  
mydocbase.password=d7d1d6e383d6d4e1d0  
mydocbase.domain=
```

To set up J2EE principals:

To enable J2EE principals to log in to repositories (single login), you must modify the deployment descriptor file (`/WEB-INF/web.xml`) and follow the procedures that are specific to your J2EE server.

1. In `/WEB-INF/web.xml`, remove the comments around the security constraints element. This sets up a user role called "everyone". The `web-resource-name` value should match the context name of the Web application. For example:

```
<security-constraint>  
  <web-resource-collection>  
    <web-resource-name>Webtop</web-resource-name>  
    <url-pattern>/*</url-pattern>  
    <http-method>POST</http-method>  
    <http-method>GET</http-method>  
  </web-resource-collection>  
  <auth-constraint> <role-name>everyone</role-name>  
</auth-constraint>  
</security-constraint>  
  
<login-config>  
  <auth-method>BASIC</auth-method>  
</login-config>
```

2. Follow the J2EE server procedure for setting up J2EE principals. Each J2EE server has its own procedure. For example, in WebLogic, you can use the management tool to create users with names that are Documentum logins. You can set the password to anything, and the password does not have to match the Documentum password. In Tomcat, you specify the J2EE principals in a configuration file, `/conf/tomcat-users.xml`.

Single sign-on

Content Server version 5 and higher supports pluggable authentication or single sign-on (SSO). You must also set up a SiteMinder realm to perform authentication for WDK applications. The `dm_netegrity` plugin installed in the Content Server decodes the `SMSESSION` token sent from WDK for authentication. The plugin contacts the Netegrity policy server to verify that the token is valid. Errors in authentication are logged in the `/Documentum/dba/log/dm_netegrity.log` file.

To summarize, perform the following steps to enable single sign-on in a WDK-based application:

1. Configure SiteMinder to authentication Documentum users. (Refer to the Netegrity SiteMinder documentation.)
2. Configure the Web application server to use an external HTTP Server supported by Netegrity SiteMinder. (Refer to the SiteMinder documentation.)
3. Configure the Content Server plugin. (Refer to the Server documentation.) Since the plugin is considered a custom Web Agent, there are no additional Web Agents needed for WDK.
4. Configure the WDK-based application (`app.xml` settings), described in this topic.

The single sign-on (SSO) authentication scheme `SSOAuthenticationScheme` is registered in the authentication properties file `com.documentum.web.formext.session.AuthenticationScheme.properties` located in `/WEB-INF/classes`. You must modify the properties file to make the SSO authentication scheme first in the list of authentications that are attempted during login. If the Docbase Login scheme is listed before the SSO scheme, the user will be presented with a login screen instead of single sign-on.

The WDK SSO Authentication Scheme needs three pieces of information in order to authenticate an HTTP session against a docbase:

- The name of the Authentication Plugin (`dm_netegrity`) that is used in the content server.
- The name of the ticket that will be retrieved from a vendor-specific (SiteMinder) cookie.
- User name, which is retrieved from a vendor-specific (SiteMinder) HTTP request's header or remote user.

Edit the `app.xml` file in your application's `/custom` directory. Update the element `<sso>` under the existing `<authentication>` element as follows, replacing the repository name in the `<docbase>` element. (These elements are described in detail in [Table 2-7, page 64](#), especially the possible values of `<user_header>`.)

```
<authentication>
  <domain/>
  <docbase>repository_name</docbase>
```

```
<service_class>
  com.documentum.web.formext.session.AuthenticationService
</service_class>
<sso_config>
  <ecs_plug_in>dm_netegrity</ecs_plug_in>
  <ticket_cookie>SMSESSION</ticket_cookie>
  <user_header>SM-USER</user_header>
</sso_config>
</authentication>
```

Note: Restart the application server so that the application setting changes may be picked up.

The login process is described below in the order that it occurs:

1. User issues URL request to the web application and is presented with the WDK login component, displaying only the repository chooser control.
2. Login component interrogates the HTTP request, checking for the additional Netegrity information (smsession cookie and the sm-user header values). When these are found, SSO component processing continues.
3. The login component performs the authentication with the following values created from smsession cookie and sm-user header values:
 - userName
Value in the sm-user header
 - userPassword
Plugin name concatenated with the smsession cookie value
 - docbase
Repository selected by the user in the modified login component display
4. Field contents are bound to an IDfSessionManager instance and a session becomes available to the application/user.

Ticketed login

A Web application that already has a Documentum session can link to a WDK-based application using a ticketed login. The ticket logs the user in without a login screen, because the user is already logged in through the calling application.

The link (URL) containing a ticketed login is in the following form, with spaces escaped:

```
http:// server_name:port_number/ application_name/component/
component_name?
  locale= locale_code&ticket= DM_TICKET%3d0000001a3dd7626e. docbase_name@ host_name&
  username= username&docbase= docbase_name
```

Key:

- *server_name:port_number*
Host-specific alias for accessing the server
- *application_name*
Virtual name for your application, used to access the application
- *component= component_name*
Redirects to a specific component. You can also redirect to an action by substituting *action= action_name*. Specifies a specific component to be launched. If redirecting to an action, specify action name.
- *locale= locale_code*
Sets the locale for the session using Java locale code. Localized strings for that locale must be present.
- *ticket= ticket number*
Specifies a ticket that has been generated within the required time frame (default 5 minutes) generated by the DFC call `getLoginTicket()` or `getLoginTicketEx()`
- *docbase_name*
Target repository, appended to the ticket with "."
- *host_name*
WDK application server host name
- *docbase= docbase_name*
Name of target repository

For example, the following URL contains a login ticket (line break inserted for display purposes only):

```
http://localhost:8080/webtop/component/main?locale=en_US&ticket=
  DM_TICKET%3d0000001a3dd7626e.viper@
  denga000&username=randy&docbase=viper
```

Note: Arguments must escape single quotes as `%27` and embedded equal signs as `%3d`. The ticket argument in the example above, before escapes, is `DM_TICKET=0000001a3dd7626e.viper@denga0008`. The argument after escapes is `DM_TICKET%3d0000001a3dd7626e.viper@denga0008`

The URL can have an optional `startupAction` parameter so that the action is called after the ticketed login. If you specify a startup action, you must also provide required action arguments in the URL.

In the following example, the startup action arguments are provided, with all spaces and embedded equal signs escaped.

```
http://localhost:8080/webtop/component/main?startupAction=
  search&query=select%20object_name%20from%20dm_document
  %20where%20r_object_id%3d%2709aac6c2800015b7%27
```

```
&queryType=dql&ticket=DM_TICKET%3d0000001a3dd7626e.viper  
&denga0008&username=testuser&docbase=viper
```

The ticket is generated by an API call and expires by default in 5 minutes. The ticket expiration time can be set in the `login_ticket_timeout` attribute of the content server configuration object. Your code should generate a new ticket every time a user clicks on the link that launches the WDK-based application.

To get a login ticket for a user who is currently logged in, use DFC calls similar to the following. (The class that encodes embedded characters to make them URL-safe is `java.net.URLEncoder`).

```
IDfSession sess = null;  
try  
{  
    IDfSessionManager sessionManager =  
        SessionManagerHttpBinding.getSessionManager();  
    sess = sessionManager.getSession(strDocbase);  
    String strPrefix = "http://localhost/wtapp/  
        component/main?ticket=";  
    String ticket = sess.getLoginTicket();  
    String strSuffix = "&username=myname&docbase=mydocbase";  
    String fullUrl = strPrefix + URLEncoder.encode(ticket) + strSuffix;  
    System.out.println(fullUrl);  
}  
finally  
{  
    if(sess != null)  
    {  
        releaseSession(sess);  
    }  
}
```

Skip authentication

All components automatically call the login dialog if the user does not have a session. If your custom component does not require a Documentum session, you can configure skip authentication for the component. Skip authentication is configured in the resource file `Environment.properties`, which is located in the directory `/WEB-INF/classes/com/documentum/web/formext`. To add a component that skips authentication, add a line with the key value `non_docbase_component`. In the following example, the custom component `bluesheet` does not require a Documentum session:

```
non_docbase_component.6=bluesheet
```

You can use JSP pages and server-side classes from your JSP pages that do not require a repository connection. Do not use these pages within a component so that the login dialog is not called.

Explicit login

You can launch the login component explicitly using the standard component URL:
`/component/login`.

After successful explicit login, the login dialog will forward to either a component or a URL that is specified in the login component definition.

To navigate to a given component after login, type a URL in the following form:

```
/my_app/component/login?entryComponent=acme
```

To navigate to a given component and an entry page that is named in the component definition, type a URL in the following form:

```
/my_app/component/login?entryComponent=acme&entryPage=welcome
```

To navigate to any URL after login, type a URL in the form:

```
/my_app/component/login?entryUrl=/acme/index.jsp
```

Login preferences

The login component uses the WDK Preferences service to store the following login settings in a cookie:

- user name
- repository
- domain
- language
- show options flag (default true)

When a user changes one of these settings, the new setting is written to the preference store. If the user has never chosen a repository or domain, the advanced options are shown regardless of the login component `showOptions` configuration value.

If the user selects a value (such as a repository) that is valid for one server but not for another server, the preference is ignored, and the default value for the server is presented in the login dialog. The user may then select another value through the UI.

Login locale

The initial locale for the UI presentation at login is determined by the locale of the J2EE server host. The login component presents a language dropdown control that lists all of the installed locales for the application. When the user selects the locale, the UI is refreshed with the UI strings of the selected locale.

Number of user sessions

To set the maximum number of application server sessions, specify an integer value in the `<session_config>.<max_sessions>` element of your custom `app.xml` file. After the maximum number of sessions has been reached, requests are redirected to the JSP page `/wdk/serverBusy.jsp`. A value of `-1` means that there is no limit to the number of sessions.

Using events and JavaScript

An event calls specific application code. Controls raise events on the client, and you can configure the events to be handled on the client or server. Server events are raised in server-side code and can be handled on the client or server.

Server event handling provides code reuse across the application, state management, and better performance. Client-side event handling is also supported, to enable dynamic form behavior, standard HTML events such as `body onload()`, reduced round-trips to the server, and frameset processing.

Server events are raised by controls in a JSP page and are handled in the component class that owns the JSP page. For a description of control events and event handling, refer to [Control events, page 165](#).

The following information is available on client-side events:

- [Navigating with an event handler, page 112](#)
- [Client-side navigation, page 113](#)
- [Registering client event handlers, page 114](#)
- [Using client-side scripts, page 115](#)
- [Events between frames, page 117](#)
- [Managing frames, page 120](#)
- [Calling JavaScript functions from server-side classes, page 121](#)

Navigating with an event handler

In a JavaScript event handler, you can navigate to a component from within a JSP page. For example, a button in the about component UI sets the `runatclient` attribute to `true`. The button's `onclick` eventhandler is specified as `onIAPIEditor`. The event handler is in the same JSP page, and it navigates to the API component as follows:

```
function onIAPIEditor()  
{  
    postComponentNestEvent(null, "api", "api");  
}
```



```
}

```

Components are addressed by URLs to the component dispatcher servlet. The syntax of the URL that calls a component is:

```
/ app/component/ component_name/ [/ page_name] [ ? params ]
```

where:

- *app*: Deployed application root context directory.
- *component_name*: Name of the component defined in the component definition XML file.
- *page_name*: Logical page name defined in the component definition XML file. If not present, the page defined by the <start> element is used.
- *params*: (Optional) Scope parameter and value pairs. If there are scoped definitions for the component, the parameters specified in the URL are used to dispatch the appropriate component definition.

Following are two examples of URLs to components:

```
/wp/component/publish?objectId=xxx
```

```
/webtop/component/properties?objectId=yyy
```

A JavaScript function that calls a component is shown below:

```
function onClickDQL()
{
    newwindow = window.open(
        "/" + getVirtualDir() + "/component/dqleditor", "dqleditor",
        "location=no,status=no,menubar=no,toolbar=no,resizable= yes,scrollbars=yes");
    newwindow.focus();
}
```

Note: URLs in JSP pages must have paths relative to the Web application root context or relative to the current directory. For example, the included file <%@ include file='doclist_thumbnail_body.jsp' %> is in the same directory as the including file. The included file <%@ include file='/webcomponent/navigation/drilldown/drilldown_body.jsp' %> is in the /webcomponent subdirectory of the Web application.

Client-side navigation

Use client-side navigation functions when you need to handle a client-side event by nesting or jumping to another component. The JavaScript file /wdk/include/componentnavigation.js contains the following client-side component navigation methods:

- `postComponentJumpEvent()`: Jumps to another component. For example, in Webtop the page `tabbar.jsp` contains a JavaScript function that calls `postComponentJumpEvent()`. The parameters are the form ID (can be null), source

component, (optional), target frame, (optional) event name, and (optional) event argument. For example, in the Webtop page titlebar.jsp:

```
function onSearch()
{
    postComponentJumpEvent(null, "search", "content");
}
```

To open the new component in the same frame, use "_self" for the target frame parameter.

- `postComponentNestEvent()`: Nests to another component. This function has the same arguments as `postComponentJumpEvent()`, above. For example, in Webtop classic.jsp:

```
function
authenticate(docbase)
{
    // nest the modal authentication dialog ready for login
    postComponentNestEvent(null, "authenticate", "content", "docbase", docbase);
}
```

Registering client event handlers

Event handler registration provides control over where events are handled. WDK provides a `registerClientEventHandler()` method to register client event handlers. This script is automatically included in all pages rendered to the browser. The signature of the method is:

```
function registerClientEventHandler(strSrcWindowName, strEventName,
    fnEventHandler);
```

where:

- `strSrcWindowName`: String (optional) source frame or window name
- `strEventName`: String event name
- `fnEventHandler`: The event handler function pointer

If the source frame or window name is `NULL`, the event is handled by any parent window in the hierarchy regardless of which frame fired it. If the frame name is not null, the event is handled only if it was fired from the specified frame.

The following example registers an event handler for the event "treeNodeSelected" that is fired from the tree frame. The event handler is registered in the parent JSP page to handle the event named `treeNodeSelected` when the event is fired from the tree frame. The registration sets `onNodeSelected` to handle the event:

```
<script>
    registerClientEventHandler("tree", "treeNodeSelected", onNodeSelected);
    function onNodeSelected(nodeId)
    { ... }
</script>
```

```
<frameset cols="50%,50%">
  <frame name="tree" src="tree.jsp">
  <frame name="list" src="list.jsp">
</frameset>
```

Using client-side scripts

You can include client-side scripts such as JavaScript or VBScript (supported by IE browsers only). There are two ways to include scripts in WDK: manually and registered. Script usage is described in the following topics:

- [Manual scripts, page 115](#)
- [Registered scripts, page 115](#)
- [WDK scripts, page 116](#)
- [Generated script tags, page 116](#)
- [JavaScript tracing, page 117](#)

Manual scripts

JavaScript functions can be manually included within the HTML elements of a JSP page. The following example includes the script and calls a method from the body onload event:

```
<head>
  <script src="/myapp/scripts/calendar.js" language="javascript1.2">
  </script>
</head>
<body onload="initCalendar()">
```

Registered scripts

Registered scripts are automatically inserted into every rendered WDK form by the WDK framework. Use registered scripts to provide infrastructure and behavior that is reused across the application. Scripts are registered in a Java properties file: `com.documentum.web.form.WebformScripts.properties`. Each registry entry has the following syntax:

```
index_name.property=value
where:
```

- `index`: Order of inclusion in the HTML form
- `name`: Logical name of the script

- `property`: Type of property. Valid values are `href`, `language`, and `trace`.
- `value`: Value of the property. Valid values are a URL for the `href` property, a scripting language name for the `language` property, and `true` or `false` for the `trace` property (enables script tracing).
- `forceinclude`: Set to `true` to force inclusion of the `href`, required for a portal environment

In the following example, the first JavaScript file that supports the help system will load before the second JavaScript file:

```
7_Help.href=/wdk/include/help.js
7_Help.language=javascript1.2 8_Help.href=/wdk/include/modal.js
8_Help.language=javascript1.2
```

WDK scripts

WDK registers its own scripts to support the client-side infrastructure. Do not modify the script registry entries for the WDK scripts:

- `trace.js`: Enables client-side tracing
- `locate.js`: Locates browser window frames
- `events.js`: Enables client-side events
- `scroll.js`: Stores and retrieves the scroll position
- `formnavigation.js` and `componentnavigation.js`: Enables navigation between forms and components
- `framenavigation.js`: Enables frame and browser identification by assigning IDs to frames and browser windows, and sets an in-memory cookie to identify the browser window
- `help.js`: Enables the online help to launch in a browser window
- `modal.js`: Enables modal dialogs
- `contenttransfer.js`: Registers an event handler for HTTP download

Generated script tags

Script tags are generated by the WDK framework when a `<dmf:webform>` tag is rendered. The generated script tag is similar to the following:

```
<script>
  var Trace_CLIENTEVENTS=true;
  var Trace_CALENDAR=false;
</script>
<script src="/myapp/include/calendar.js" language="javascript1.2"/>
```

The script variable (e.g. Trace_CLIENTEVENT) is taken from the script entry name. The src attribute value of the script tag is taken from the script property href. The language attribute value of the script tag is taken from the script property language. Any valid HTML language value may be specified in the script registry

To generate script tags, place the <dmf:webform> tag within the head elements of a JSP page. Many WDK controls are dependent on the scripts that are included by the scripts registry. The <dmf:webform> tag also invokes the form processor before any other tags are processed.

JavaScript tracing

Script tracing — If script tracing is enabled (refer to [Client-side tracing, page 342](#)), a separate browser window opens with the WDK application, and client-side tracing messages are sent to the tracing window while the application executes. To add tracing messages to your client-side script, use the following syntax:

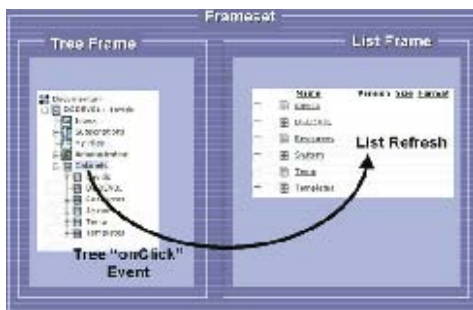
```
if (Trace_ ScriptName)
{
    Trace_println("tracing message here");
}
```

For example:

```
if (Trace_Calendar)
{
    Trace_println("Calendar initialized");
}
```

Events between frames

Client-side processing is often required to synchronize the content of each frame in the application frameset to reflect user interaction with the application. Typically, inter-frame events are fired by client-side control event handlers. The control sets the runatclient attribute to true to specify that the event is handled on the client. The event handler is specified as the value of the onclick or onselect attribute. This is illustrated by a folder browser. In the following example, two frames are used: One contains a folder tree, and the other contains a list of objects that exist in the selected folder of the folder tree. The tree frame contains a tree control that fires the onclick event. In order to propagate the onclick event to the list frame, a client-side onclick event handler is used.

Figure 2-2. Folder tree frame interaction

If your event handler is not in a parent window of the frame in which the event is fired, the event will not be handled. You can control where the event is handled by registering the event handler. (Refer to [Registering client event handlers](#), page 114.)

WDK JSP pages can fire an inter-frame event using the `fireClientEvent()` function. This function is defined in the JavaScript file `events.js`. This scrip file is automatically included in all rendered HTML pages. The `fireClientEvent()` function informs other pages of the event. The signature of this method is:

```
function fireClientEvent(strEventName);
```

where `strEventName` is a String representing the event to fire. You can specify additional parameters which are then passed on to the event handler as event parameters. All arguments that are given to `fireClientEvent` are passed automatically to the registered event handler.

Example 2-1. Firing an inter-frame event

In the following example from Webtop `tabbar.jsp`, the `onComponentSelect` event is fired when the user clicks a tab, and the component for the selected tab is passed as an event argument:

```
function onClickTab(component)
{
    refreshCurrentComponent(component); fireClientEvent(
        "onComponentSelect", component);
}
```

The JSP page `streamline.jsp` has registered an event handler for this event:

```
registerClientEventHandler(
    "tabbar", "onComponentSelect", onComponentSelect);
```

The handler processes the event arguments and sets the current component:

```
function onComponentSelect(component)
{
    g_strCurrentComponent = component;
}
```

Inter-frame event handlers

Events are propagated through the client-side window hierarchy. When `fireClientEvent()` is called, the framework searches for windows that contain an associated event handler. The window that fired the event is checked first, followed by a recursive pattern where the parent window is visited until the root window is reached. Forms are thus encapsulated and can be reused in other applications without pulling in dependent forms.

For control over the location of event handlers, register the event handler. Refer to [Registering client event handlers, page 114](#) for more information.

Inter-frame server events

Inter-frame event handlers can post server-side events using `postServerEvent()` (refer to [Firing a server event from the client, page 417](#)). This allows the behavior or layout of one form to be synchronized with the user input or other operation in another form.

Example 2-2. Using `postServerEvent()` to trigger container navigation

In the following example, a JSP page loads and checks whether applets are installed. If they are not, a server event is posted that is handled by the container class. The container class jumps to another JSP page.

In the webcomponent page `checkApplet.jsp`, a server event is posted in the JavaScript `finish()` function (which is called in the body onload event):

```
function finish()
{
    var bInstalled =
    checkContentXferAppletInstall(
        "<%=IContentXferConstants.VERSION_NUMBER%>", false,
        <%=bIsPlugInBeingUsed%>);

    if (bInstalled == true)
    {
        postServerEvent(null, null, null, "onCheckAppletComplete");
    }
    else
    {
        postServerEvent(null, null, null, "onNeedInstall");
    }
}
```

If the user does not have the applets installed, the `onNeedInstall` event is posted. This event is handled in the class `ContentTransferContainer`:

```
public void onNeedInstall(Control control, ArgumentList arg)
{
    setComponentPage("installapplet");
}
```

```
}
```

This method `onNeedInstall()` jumps to the component page named `installapplet`, defined in the `contentxfercontainer` component definition as `/webcomponent/library/contentxfer/installContentXfer.jsp`.

Managing frames

Frames in a WDK application are given frame IDs and browser IDs to preserve browser history and state. To take advantage of frame history and preserve memory usage on the J2EE server, use the `<dmf:frameset>` and `<dmf:frame>` tags from the WDK `dmf:form` library to generate framesets.

For information on client-side, server-side, and inter-frame events and JavaScript functions in the WDK framework, refer to [Using events and JavaScript, page 112](#).

The `<dmf:frameset>` and `<dmf:frame>` tags generate framesets in which each browser window is assigned a browser ID and each frame is assigned a frame ID. These IDs are assigned by the JavaScript functions in `framenavigation.js`. (This JavaScript file is included in all JSP pages that have the `<dmf:webform/>` tag.)

A frame ID is static and is bound to the frame. The browser ID is generated for the top frame the first time the frame loads. The combination of frame ID and browser ID allows the application to maintain browser history for more than one browser window sharing the same frameset.

The `<dmf:frameset>` and `<dmf:frame>` tags generate HTML similar to the following. Source:

```
<dmf:webform/><dmf:frameset rows="0,38,*,30">
  <dmf:frame name="titlebar" src="/component/titlebar">
  </dmf:frame>
  <dmf:frame name="status" src="/webtop/status/status.jsp">
  </dmf:frame>
</dmf:frameset>
```

Generated HTML. On the first request to the page containing the frameset, the browser ID is set as an in-memory cookie:

```
<script language="javascript1.2">
  setCookie('__dmfClientId', getBrowserId(), null, '/');
</script>
<frameset rows="0,38,*,30">
  <frame name="titlebar" id="Main_titlebar_0"
    src="/webtop/component/titlebar?
    Reload=1051237917848&__dmfFrameId=Main_titlebar_0"></frame>
  <frame name="status" id="Main_status_0"
    src="/webtop/webtop/status/status.jsp?
    Reload=1051237917848__dmfFrameId=Main_status_0"></frame>
</frameset>
```


The configurable attributes for the frame and frameset controls are described in *Web Development Kit Reference Guide*.

For navigation from one frame to another, you should use the JavaScript function `changeFrameLocationInFrameset()`. This function will ensure that the frame and browser IDs are appended to the URL. In the following example, the client-side (JavaScript) event handler calls the navigation function to accomplish a `frame.location.replace()` navigation:

```
function onStreamlineView(view)
{
    changeFrameLocationInFrameset( parent.parent, "view",
        "<%=request.getContextPath()%>/webtop/streamline/streamline.jsp");
    ...
}
```

To call a component in another frame, use the JavaScript function `postComponentNestEvent()`. For example, the Advanced Search link in Webtop's `titlebar.jsp` specifies that the advanced search component will load in the content frame (the third argument):

```
function onClickAdvancedSearch()
{
    var contentPage = eval(getAbsoluteFramePath("content"));
    if (contentPage != null)
    {
        var textField = document.getElementById("txtSearch");
        var strValue = textField.value;
        ...
        postComponentNestEvent (null, "advsearchcontainer", "content", "
            component", "advsearch", "type", "dm_sysobject", "basetype", "
            dm_sysobject", "usepreviousinput", "false", "query", strValue );
        ...
    }
}
```

Calling JavaScript functions from server-side classes

Your JavaScript function may require information from the component class. For example, a body onload event calls a server-side event handler, perform some computation, and return a value. You need to get the value in another JavaScript function:

```
<script>
function invokeComputation()
{
    postServerEvent (null, null, null, "onComputation");
}
</script><body onload='invokeComputation()' ...>
```

In your component class, you pass the required information back to the client by calling `setClientEvent()`. Do not encode client event arguments using

`SafeHTMLString.escapeText()`. Instead, use `SafeHTMLString.escapeScriptLiteral()` to encode client arguments.

```
public void onComputation()
{
    //do computation
    ArgumentList arg = new ArgumentList;
    arg.add("variable1", value1); //your return value
    setClientEvent("getComputation", arg);
}
```

In your JSP file, the named client event gets the argument value:

```
<script>
function getComputation(arg)
{
    alert(arg);
}
```

You can also call `setClientEvent()` from an action execution method. Then you handle the client event in the component JSP page from which the action was launched. Do not encode client event arguments using `SafeHTMLString.escapeText()`. Instead, use `SafeHTMLString.escapeScriptLiteral()` to encode event arguments.

Branding an application

The branding service allows you to customize the look of your application user interface (UI) by incorporating colors, fonts and images. The branding service manages the UI appearance using themes, which incorporate images, icons, and cascading style sheets (CSS). Resource files for your themes are organized into resource directories.

The default portal theme is iconic, which allows the portlets to use the portal styles.

The information about themes and tasks to extend or create a theme include:

- [Registering a theme, page 123](#)
- [Creating a theme directory, page 124](#)
- [Making a theme available, page 125](#)
- [How themes are located, page 126](#)
- [Using style sheets, page 127](#)
- [Identifying styles in WDK applications, page 130](#)
- [Adding images and icons, page 133](#)
- [Configuring buttons, page 134](#)
- [Configuring the file selector applet, page 135](#)
- [Branding examples, page 135](#)

The example in the topics listed above sets up a new theme named *topteam*. The example covers all of the tasks required to create the new theme.

Registering a theme

Branding themes are defined in the `<themes>` element in the custom application `app.xml` file. In the following example from a custom `app.xml` file, the `<themes>` element specifies a default theme, a resource bundle for theme names that are displayed in the UI, and a set of themes including a custom theme. Note that the `<nlsbundle>` element points to a custom properties file that you will create:

```
<!-- List of themes available in general preferences -->
<themes>

    <!-- Default theme to use when webtop starts up -->
    <default-theme>trendy</default-theme>

    <nlsbundle>com.documentum.custom.BrandingServiceNlsProp</nlsbundle>
    <theme>
        <name>documentum</name>
        <label><nlsid>MSG_BRAND_DOCUMENTUM</nlsid></label>
    </theme>
    ...
    <theme>
        <name>topteam</name>
        <base-theme>documentum</base-theme>
        <label><nlsid>MSG_BRAND_TOPTEAM</nlsid></label>
    </theme>
</themes>
```

Note: If your theme does not extend another theme, you do not need the `<base-theme>` element.

Inheritance occurs only at the `<themes>` primary element level, so you must copy the entire contents of the `<themes>` element from the `/wdk/app.xml` file into the `/custom/app.xml` file.

The `<themes>` element contains the following elements:

- `<default-theme>` – Defines the name of the default theme
- `<nlsbundle>` – Defines which NLS bundle to use to interpret localized strings (in this case, `BrandingServiceNlsProp` contains the mapping table). This element is optional and, if omitted, its value is inherited from `<application>`.
- `<theme>` – Defines a unique theme. The `<themes>` element can contain one or more `<theme>` elements. Each `<theme>` element contains the following elements:
 - `<name>` – Defines the unique name of the theme, as used by the `<default-theme>` element and the `<base-theme>` element. This name is not visible to users.
 - `<base-theme>` – Defines the name of the theme on which the current theme is based. By default, the current theme inherits the type icons and format icons from the directory structure of the specified base theme.
 - `<label>` – Defines the user-readable, internationalized name of the theme (that is, the text that appears in the General Preferences dialog box).

Note: After saving your changes (for example, when adding a new theme directory), you must make them known to the application by navigating to the refresh utility page `/wdk/refresh.jsp`.

Creating a theme directory

Each application layer directory contains a theme directory that contains several themes. A theme in one layer is available to the application layer and to other themes that extend that theme.

The root WDK directory (`/wdk`) contains a theme directory. The `/webcomponent` directory contains a theme directory whose themes contain additional resources that are used in the components in that application layer. For example, the luxury theme in the webcomponent layer inherits all of the luxury theme resources in `/wdk/theme` and adds a style sheet and images used by components in the webcomponent layer.

Your custom application can have its own themes directory or directories. You must register your theme in the custom application `app.xml` file (refer to [Registering a theme, page 123](#)).

Your new theme directory must contain the following subdirectories for the theme resource files:

- `/custom/theme/ theme_name/css`

This directory stores all of the CSS for the theme. In most cases, there is only one CSS per theme. If more than one CSS per theme exist, they are used in alphabetical order.

- `application_layer/theme/ theme_name/icons`

This directory stores all of the icons for the theme.

- `application_layer/theme/ theme_name/images`

This directory stores all of the images for the theme. The images are used by controls, as defined in the control attributes in each JSP.

Note: The theme directories are present in the installed application only if they contain files or other directories.

Creating a new theme with all new content — When you create a new theme, create all of the subdirectories listed above. Copy content into these directories for all of the images, icons, and styles that you will be using in your application. Copy these from the existing theme directories in the `/wdk`, `/webcomponent`, and application layer theme directories. For example, if your new `topteam` theme uses all of the icons from the `documentum` theme but doesn't extend the `documentum` theme, copy all of the files and subdirectories under `/wdk/theme/documentum/icons`, `/webcomponent/theme/documentum/icons`, and `/webtop/theme/documentum/icons` into `/custom/theme/topteam/icons`. Do the same for

the /images and /css subdirectories of the documentum theme in the application layers. Then you can substitute your own images, icons, or styles as appropriate.

Creating a theme that extends another theme — Unless you intend to create new images, icons, and styles for every component, you should extend an existing theme so that you can simply override the images, icons, and styles in that theme. Create directories only for the resources that override content in the parent theme. For example, if you are extending the trendy theme and you are using the same styles in the trendy stylesheets webforms.css, webcomponents.css, and webtop.css, then you do not need a /css directory.

Making a theme available

When you add a new theme, you must make it available in the UI by adding it to the list of themes in a properties resource file.

The following example extends the list of themes by using NLS_INCLUDES, which reads the included properties file into your custom file. Create a text file named BrandingServiceNlsProp.properties in /custom/strings/com/documentum/custom or the location that you have specified for the value of the <nlsbundle> element in the custom app.xml file.

Add the following content to the custom BrandingServiceNlsProp.properties file:

```
NLS_INCLUDES=com.documentum.web.common.BrandingServiceNlsProp
MSG_BRAND_TOPTEAM=TopTeam
```

The resulting dropdown theme list is similar to the following:

Figure 2-3. Custom Theme



How themes are located

A control (such as the OK button, label title, View tab bar, and so on) can be configured with the CSS style and the location of the image files used to render that control. The branding service searches for each referenced style and image as follows: The current theme, then the base theme, then the base theme of the base theme, and so on. In the standard installation, the base theme is the documentum theme.

The dependencies between the application layers are the same as the dependencies as specified by the extends attribute in an application layer app.xml file:

1. `wdk`: The base layer (no dependency on any other layer)
2. `webcomponent`: Dependent on the WDK layer
3. *application-specific directory*: If present, dependent on the webcomponent layer or another application-specific layer
4. `custom`: The most dependent layer, dependent on either the webcomponent layer or an application-specific layer

For example, to load `/icons/type/t_dm_document_16.gif` in the coolblue theme, the resource loader loads the first file named `/t_dm_document_16.gif` that it finds from the following search path:

```
/custom/theme/coolblue/icons/type/t_dm_document_16.gif  
/webtop/theme/coolblue/icons/type/t_dm_document_16.gif  
/webcomponent/theme/coolblue/icons/type/t_dm_document_16.gif  
/wdk/theme/coolblue/icons/type/t_dm_document_16.gif  
/custom/theme/documentum/icons/type/t_dm_document_16.gif  
/webtop/theme/documentum/icons/type/t_dm_document_16.gif  
/webcomponent/theme/documentum/icons/type/t_dm_document_16.gif  
/wdk/theme/documentum/icons/type/t_dm_document_16.gif
```

Branding is configured in the same way as other features in WDK. You should make all your changes in copies of the original files and then store your changes in the `/custom` application directory.

The branding service processes theme files as follows:

1. When a form is rendered, the branding service searches for and processes CSS files. Most applications use only one CSS but can use more than one. The branding service processes first the CSS in the base application directory and then works through the application hierarchy, processing each CSS file it finds.
2. When a form is rendered, the form contains references to resource files, which in most cases are image (graphic) files. The branding service searches for and resolves each reference into a URL for the appropriate GIF or other image file. The branding service searches first for the resource files in the most dependent application

directory (that is, the /custom directory), then works up through the application hierarchy to the base application.

Using style sheets

Each theme directory contains a /css subdirectory with styles that apply to controls in the application layer. In most cases, there is one .css file in each theme /css folder, or the folder does not exist at all. The branding service includes and renders a reference to each CSS file that exists.

All controls that use Cascading Style Sheet (CSS) classes and styles use either a default class from webform.css or a custom class that is set as a control tag attribute. The default class for a control is overridden by a class or style that you set as a control tag attribute in a JSP page. To override a style, specify the CSS style definition as the value of the style attribute on a control. In the following example, the style rule overrides the default label style defined in the CSS files:

```
<dmf:label nlsid = "LABEL_DESCRIPTION" style=style='font-family:Courier  
New,Courier;font-size:9px' />
```

You can override a style globally by defining the style in your custom CSS file, because your custom application styles overrides all other style rules with the same name.

Note: When you override a style, it still inherits rules within the style that are not overridden. You may need to override all rules within the style. For example, you wish to override the background color in following style within /app/webtop/theme/tahoe/css/webtop.css:

```
.webtopTitlebarBackground  
{  
  background-color: #ffcc33;  
  background-image: url("../images/titlebarbg.gif");  
}
```

If you override in the following way, the image will still display:

```
.webtopTitlebarBackground  
{  
  background-color: red;  
}
```

You must add a rule that overrides the image, as follows:

```
.webtopTitlebarBackground  
{  
  background-color: red;  
  background-image: url("");  
}
```

A style that is defined in the base theme (or lower application layer) can be redefined in a derived theme or a higher application layer. For example, the style .myStyle defined

in `/wdk/theme/documentum/css/webforms.css` will be replaced by the definition of `.myStyle` in `/custom/theme/topteam/css/custom.css`.

The list of style sheets to be applied to a JSP page is assembled at run time. The branding service searches theme folders in the application folder path, searching for files with a `.css` extension. The style sheets for the base theme, defined in `app.xml`, are included before the style sheets for the theme itself. For a specific theme, the branding service searches for the theme based on the application layer hierarchy. For example, if the user has selected the `topteam` theme, which extends the `documentum` theme, the service searches directories in the following order (reading from top to bottom) to render a list of style sheets:

```
/wdk/theme/documentum/css/webforms.css
/webcomponent/theme/documentum/css/webcomponents.css
/webtop/theme/documentum /css/webtop.css
/custom/theme/topteam /css/custom.css
```

The list of style sheets is rendered into HTML similar to the following:

```
<link type="text/css" rel="stylesheet"
  href="/webtop/wdk/theme/documentum/css/webforms.css">

<link type="text/css" rel="stylesheet"
  href="/webtop/webcomponent/theme/documentum/css/webcomponents.css">

<link type="text/css" rel="stylesheet"
  href="/webtop/webtop/theme/documentum/css/webtop.css">

<link type="text/css" rel="stylesheet"
  href="/webtop/custom/theme/topteam/css/custom.css">
```

Style sheets in the same directory are added to the list in alphabetical order. For example, if a directory `/custom/topteam/css` contains `custom_a.css` and `custom_b.css`, the `custom_b.css` file will be listed second, and styles in the second CSS file will override styles with the first CSS file. The last definition encountered for a style is used by the browser.

Using images in style sheets

Images referenced within style sheets — Images can be referenced within a style sheet using relative paths. For example:

```
.drilldownHeader { BACKGROUND-COLOR:
  transparent; BACKGROUND-IMAGE:
  url('../images/streamline/tabbarbg.gif') }
```

If you customize an image that is referenced within a style sheet, the style sheet that references the image must be located in your custom folder so that the correct image is used.

Default WDK style sheet

The following table lists the WDK-specific styles in the WDK layer themes. The table does not define general HTML styles, such as TD or TH; for details of their usage, refer to a CSS or HTML reference. Several styles with similar application are grouped together with the notation *style_name.XXX*.

Table 2-43. webforms.css (WDK layer)

Style	Description
.accessiblemenustyle	Sets the style on the text for menu icons that display strings for accessibility (image accessibility strings). (Refer to Image accessibility strings, page 587.)
.action.XXX	Used by drilldown, searchresults and searchresultslist to render the action button.
.buttonLink .buttonDisabledLink	Sets styles on buttons that link to a URL.
.contentBackground	Sets the background color for a content frame.
.contentBorder	Sets the color of the containers (the border area around the contained components).
.databoundXXX	Used by the examples to set styles on databound controls.
.defaultXXX	Sets default styles for controls that do not have style or cssclass attribute values set on the JSP page.
.dialogXXX	Sets the style for the display of the dialog title, file name, and details that are displayed in a dialog.
.elementBackground .elementText	Sets background and text for samples.
.errorMessageLabel	Sets the style for the label on an error message.
.headerBackground	Sets the background for a table header.
.menuXXX	Sets styles for menu items.
.pagerBackground	Sets the background for a pager control.
.rowSeparator	Sets the color of the row separator lines used in the datagrids of the object list, resultlist, search results, messages and drilldown.
.validatorMessageStyle	Sets the style of the text in a validator error message.

Internationalized style sheet

The Documentum theme contains an internationalized style sheet that specifies a Unicode font, Arial Unicode MS. You can install this font on Windows client systems to support any language that can be encoded in Unicode.

Modifying a style sheet

You can modify style sheets to change the look of applications. Style sheets override styles defined in application layers that are extended by the top application layer. For example:

- The style sheet in the `/webtop/shiny/css` directory overrides the style sheet in the `/webcomponent/shiny/css` directory.
- The style sheet in the `/custom/shiny/css` directory overrides the style sheet in the `webtop/shiny/css` directory.

To redefine a control style:

1. Inspect the JSP page or view the page source to find the name of the css class that is used by a control. For example:

```
<dmf:label nlsid = "LABEL_DESCRIPTION" cssclass=  
  'defaultDocbaseAttributeStyle' />
```

2. Create a new definition of the style and write it to the `.css` file in `/custom/theme/theme name/css`. If the file does not exist, create it with any name. For example, the old class was defined in `webforms.css` as:

```
.defaultDocbaseAttributeStyle { }
```

New definition:

```
.customDocbaseAttributeStyle { FONT-SIZE: 9px }
```

3. Reference the new style name as the value of the style attribute in the JSP page. For example:

```
<dmf:label nlsid = "LABEL_DESCRIPTION" cssclass='customDocbaseAttributeStyle' />
```

Identifying styles in WDK applications

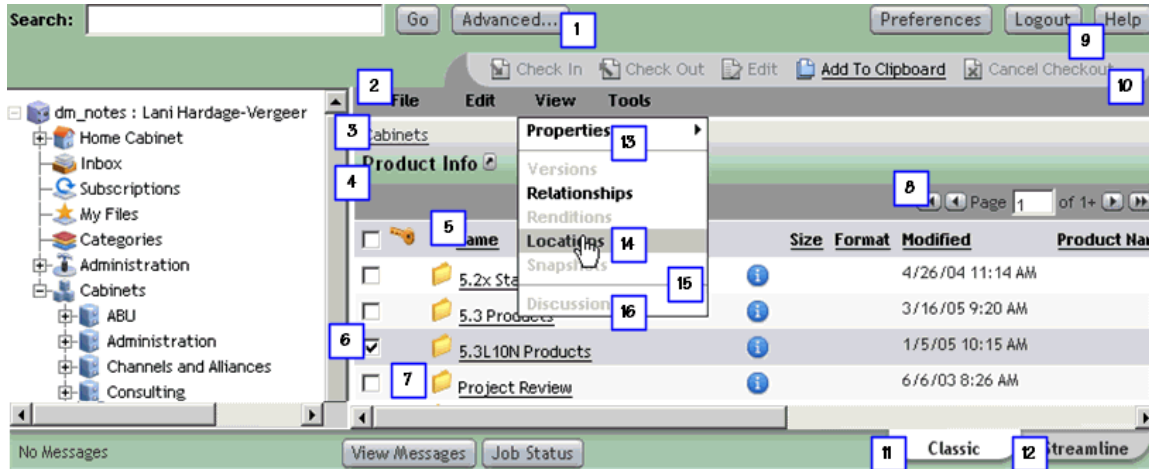
When you are creating a new theme, you want to apply your styles to all of the UI elements in your application. It can be daunting to look at the UI and try to figure out which styles in the CSS files apply to which element in the UI. The following illustrations map some of the most commonly used styles to the Webtop UI.

A simple way to locate an individual style is to find it in the UI and right-click to view the source. Search for the string that is displayed by your particular style. For example, in the illustration below you are looking for the style that is used to render the location (with the shortcut arrow next to it). If you search on the string "Product Info" in the source, you find the following HTML element with the style information (highlighted):

```
<span class='webcomponentTitle'>Product Info</span>
```

The styles in the Webtop classic view are shown below.

Figure 2-4. Webtop classic view styles



Key: Source of class is /wdk/theme/theme_name/css/webforms.css (WDK), /webcomponent/theme/theme_name/css/webcomponents.css (WCL), or /webtop/theme/theme_name/css/webforms.css (WT).

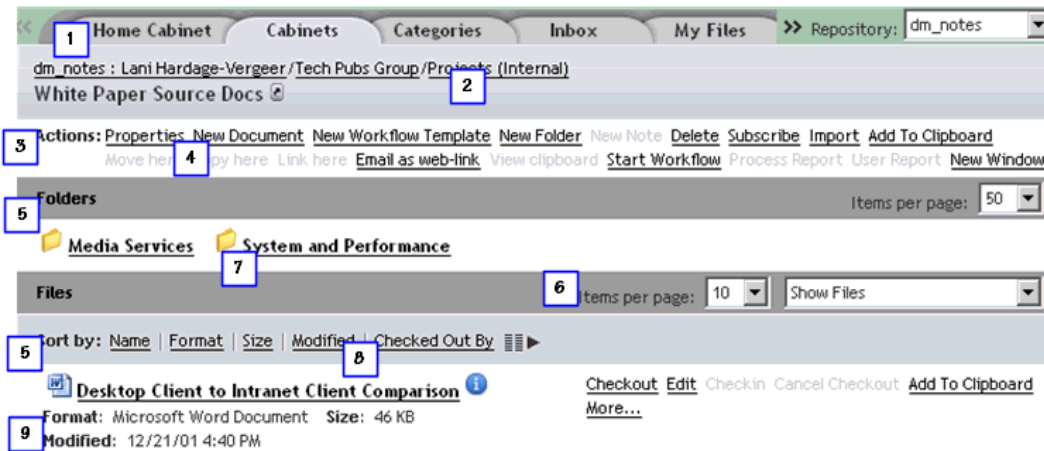
Table 2-44. Styles in Webtop classic view and menus

UI Element	Source	Style
1. Button text	WDK	buttonLink
2. Toolbar text	WDK	menuBar
3. Breadcrumb	WDK	webcomponentBread-crumb
4. Location	WDK	webcomponentTitle
5. Column header	WDK	doclistbodyDatasortLink <tr> class colHeaderBack-ground
6. Checkboxes	WDK	doclistcheckbox

UI Element	Source	Style
7. Row colors	WDK	White: <tr> class default-DatagridRowAltStyle Other: defaultData-gridRowStyle
8. Pagination elements	WDK	Area: pagerBackground Text: defaultLabelStyle
9. Button spacing	WDK	buttonbuffer
10. Toolbar action links	WDK	Active and disabled: toolbaractions
11. Selected tab	WT	webtopStatusbarTabbarSelectedText
12. Deselected tab	WT	webtopStatusbarTabbarNormalText
13. Menu text	WDK	menu
14. Selected item	WDK	menuHighlight
15. Horizontal separator	WDK	menuSeparator
16. Disabled menu item	WDK	menuDisabled

The styles in the Webtop streamline view are shown below. Button styles are the same as the classic view (above).

Figure 2-5. Webtop streamline view styles



Key: Source of class is `/wdk/theme/theme_name/css/webforms.css` (WDK), `/webcomponent/theme/theme_name/css/webcomponents.css` (WCL), or `/webtop/theme/theme_name/css/webforms.css` (WT).

Table 2-45. Styles in Webtop streamline view

UI Element	Source	Style
1. Tab text	WT	webtopStreamlineTab-barNormalText
2. Breadcrumb text	None, defaults to browser text font	defaultBreadcrumbStyle
3. Actions title	WDK	defaultLabelStyle
4. Actions text	WDK	actions
5. Section header	WCL	drilldownTitle
6. Pagination elements	WDK	defaultLabelStyle
7. Folder names	WCL	drilldownDirectoryName
8. Sorting links	WDK	defaultDataSortLinkStyle
9. Bold labels	WCL	drilldownLabel
10. Horizontal rule (bottom, not shown)	WDK/.../dragdrop.css	Tdnd, 0dnd

Drag and drop regions and effects, whose styles are named `*dnd*`, are configured in `/wdk/theme/.../css/dragdrop.css`.

Adding images and icons

Image and icon directories contain GIF files that are used to draw the control. Most controls specify their image names as `left.gif`, `bg.gif`, and `right.gif` (where `bg.gif` is the background image).

Note: Store your customized images and icons in the custom application directory.

If you are adding an icon to be displayed for custom objects or folder types, place a 16x16 pixel icon in the `/custom/theme_name/icons/type` where `theme_name` is the theme for which you wish the icon to be used. The icon file must be named with the custom object type name and with a `t_` prefix and a `_16` postfix. For example, if your custom type is named `my_sop`, the icon would be named `t_my_sop_16.gif`.

If you are providing an icon to replace one of the default application icons, the directory path below `/custom/theme/theme-name` must be the same as the original. For example, if

you are adding custom images for the paging controls, you add images named `first.gif`, `last.gif`, `next.gif`, and `previous.gif` to the directory `/custom/theme/topteam/images/paging`.

A custom image file must have the same name as the file that is used by the control in other themes. For example, if you use your own images for the paging controls, you must provide images named `first.gif`, `last.gif`, `next.gif`, and `previous.gif`. If the type or format is not databound or an image is not found, the icon resolves to `t_unknown_16.gif` or `t_unknown_32.gif`.

Note: Make sure that your customized images and icons have exactly the same dimensions as the originals. Images that do not have the same dimensions will have unpredictable effects on the UI.

Images can be referenced within style sheets. For details of how to configure such images, refer to [Modifying a style sheet, page 130](#).

Accessibility — All graphics in the `/images` and `/icons` directories must have an entry in an accessibility resource file to support accessibility. The NLS string is displayed as an HTML alt attribute value in browser mouseover. Refer to [Image accessibility strings, page 587](#) for more information.

Configuring buttons

To form the background for a text label, button, or other type of image, the control images are displayed in order from left to right. Any text that you specify in the form of an `nlsid` attribute is displayed on top of the center image (`bg.gif`), and the width of this image expands to display the entire text. The style of the text or link that is displayed is determined by the `cssclass` attribute on the JSP tag.

For example, the clipboard page `clipboard.jsp` in `/webcomponent/environment/clipboard` contains a button identified by the `nlsid` `MSG_REMOVE`. The path to the button image is specified by the `imagefolder` value, relative to the theme directory: `images/dialogbutton`. To replace the images for the Remove button, you can replace it either in one JSP page or in your entire application.

To change the text for a button:

1. The text for a button is specified by the `nlsid` attribute on the button control. Find the `nlsid` attribute for the button you want to change on the JSP page. For example, the Help button in the login page `login.jsp` has an attribute value of `MSG_HELP`:

```
<dmf:button cssclass='buttonLink' nlsid='MSG_HELP'.../>
```

2. Extend the component and override the definition for the `nlsbundle` in the extended component definition. For example:

```
<component id="login" extends=
  "login:/wdk/config/login_component.xml">
```

```
<nlsbundle>com.documentum.web.formext.session.LoginNlsProp
</nlsbundle>
```

3. Find the properties file. In the same example, the file `com.documentum.web.formext.session.LoginNlsProp.properties` is located in the directory `/wdk/strings/com/documentum/web/formext/session`.
4. Find the string identified by the `nlsid` key value on the control tag. In the same example, the key value is `MSG_HELP=Help`.
5. Create a new properties file named `MyLoginNlsProp.properties` in the directory `/custom/strings/com/documentum/custom`. Include the parent strings resource and override the button text value. In the same example, you have the following content:

```
NLS_INCLUDES=com.documentum.web.formext.session.LoginNlsProp
MSG_HELP=Resources
```

To convert a button, tab bar, or label control to an image button:

1. Specify the `imagefolder` attribute in the JSP tag.
2. In the specified image directory, put the following three files: `left.gif`, `right.gif`, and `bg.gif`. If any of these image files are not present, the control renders a generic HTML button. The tab bar control requires more than three image files; refer to the files in `/wdk/theme/theme-name/images/tabbar`.

Configuring the file selector applet

The file selector applet in the import UI (`importFileSelection.jsp`) is configured in a file named `fileSelectorAppletStyle.properties`. A copy of this file is found in each theme directory of the WDK application layers. You can set four styles: text style, size, and color, and background color. The settings for a given theme should match the text and background settings for dialog buttons in the same theme.

Branding examples

Example 2-3. Change a style definition for theme "trendy"

Write a stylesheet that redefines the style. Save it as `/custom/theme/trendy/css/newstyle.css`

The generated CSS include list might be something like: `/wdk/theme/documentum/css/wdk.css`, `/wdk/theme/trendy/css/wdk.css`, `/custom/theme/trendy/css/newstyle.css`. The new style is added after the others so it overwrites the out of the box style definitions.

Example 2-4. Add an icon for a new docbase type to theme "trendy"

Design the icon. Save large and small icons to /custom/theme/trendy/icons/type as t_my_type_32.gif and t_my_type_16.gif.

The resource search path for icons/type/t_my_type_16.gif would be in the following order:

```
/custom/theme/trendy/icons/type/t_my_type_16.gif
/webtop/theme/trendy/icons/type/t_my_type_16.gif
/webcomponent/theme/trendy/icons/type/t_my_type_16.gif
/wdk/theme/trendy/icons/type/t_my_type_16.gif
/custom/theme/documentum/icons/type/t_my_type_16.gif
/webtop/theme/documentum/icons/type/t_my_type_16.gif
/webcomponent/documentum/trendy/icons/type/t_my_type_16.gif
/wdk/theme/documentum/icons/type/t_my_type_16.gif
```

The search result is cached (keyed by the theme and the resource locator) so this search is only performed once. In your style design, you may wish to add this icon to documentum theme, as all the other themes would then inherit the new icons.

Example 2-5. Add a new theme "splashy" based on theme "mellow"

1. Write stylesheets to redefine the styles. Only include the styles which have changed from mellow. Save the stylesheets as /custom/theme/splashy/css/newstyles1.css, newstyles2.css, etc
2. Design the new icons and images. The image and icon names and path within the resource folder are dictated by the controls and forms which use them.
3. Create a new strings properties file. Save the file as /custom/strings/my/BrandingServiceNlsProp.properties. The contents are as follows:

```
NLS_INCLUDES=com.documentum.web.common.BrandingServiceNlsProp
MSG_BRAND_SPLASHY=splash!
```

4. Override the <themes> element in /custom/app.xml as follows:

```
<!-- List of themes available in general preferences -->
<themes>

    <!-- Default theme to use when webtop starts up -->
    <default-theme>trendy</default-theme>

    <nlsbundle>my.BrandingServiceNlsProp</nlsbundle>
    <theme>
        <name>documentum</name>
        <label><nlsid>MSG_BRAND_DOCUMENTUM</nlsid></label>
    </theme>
    <theme>
        <name>mellow</name>
        <base-theme>documentum</base-theme>
        <label><nlsid>MSG_BRAND_MELLOW</nlsid></label>
    </theme>
```



```

<theme>
  <name>shiny</name>
  <base-theme>documentum</base-theme>
  <label><nlsid>MSG_BRAND_SHINY</nlsid></label>
</theme>
<theme>
  <name>trendy</name>
  <base-theme>documentum</base-theme>
  <label><nlsid>MSG_BRAND_TRENDY</nlsid></label>
</theme>
<theme>
  <name>splashy</name>
  <base-theme>mellow</base-theme>
  <label><nlsid>MSG_BRAND_SPLASHY</nlsid></label>
</theme>
</themes>

```

The change to the `<nlsbundle>` element picks up `my/BrandingServiceNlsProp` properties.

You can include as many of the themes in `/wdk/app.xml` as you like, so long as you do include "mellow" (which splashy is based on) and "documentum" (which mellow is based on).

Configuring and localizing strings

Attribute labels are pulled from the data dictionary. If the label strings are not displayed in their localized version, you must update the repository data dictionary with the localized version or provide a localized version in your WDK-based application.

Text strings in WDK-based applications are externalized into National Language Service (NLS) properties files that you can extend and localize. The properties files are located in the `/strings` directory of each application layer. The string files are organized by component or groups of components. For example, the strings that appear in the data paging control are externalized to the directory `/wdk/strings/com/documentum/web/form/control/databound` in the file `DataPagingNlsProp.properties`.

Locale support is specified in the application configuration file `app.xml`. When the user selects a locale, the appropriately-named set of localized strings will be used. The localized strings are contained in NLS properties files.

This section describes how to internationalize applications and modify UI strings in the following topics:

- [Adding locales, page 138](#)
- [Adding strings to properties files, page 138](#)
- [Naming properties files, page 140](#)

- [Adding localized files to your application, page 140](#)
- [Inheriting strings, page 139](#)
- [Overriding strings in the UI, page 141](#)
- [Designing for and testing internationalization, page 141](#)
- [Image accessibility strings, page 587](#)

For information on using the Locale Service and retrieving strings in your custom classes, refer to [Locale service, page 580](#).

Adding locales

The application configuration file (app.xml) lists the supported locales. For example, in your application that extends WDK's app.xml you might have:

```
<language>
  <supported_locales>
    <locale>en_GB</locale>
    <locale>en_US</locale>
    <locale>de_DE</locale>
  </supported_locales>
  <default_locale>de_DE</default_locale>
</language>
```

Each locale must have a set of properties files that are named with the appropriate naming convention (refer to [Naming properties files, page 140](#)).

Adding strings to properties files

Strings for each application layer are externalized to a /strings directory in the application layer root directory. For example, the strings for the Preferences component are externalized to files in /webcomponent/strings/com/documentum/webcomponent/environment/preferences.

Strings for each component are contained in a Java *.properties file. If a button contains a string, for example, as a label, that string is specified in the component properties file. Each properties file contains strings for a specific locale. For example, if your application supports three languages, you have three properties files for each component.

New action strings can be added to the NLS properties file for the component that fires the action. If the action appears in more than one component, create a separate actions NLS file, and include that file in the NLS resource file for each component that requires the strings.

Images have an NLS entry in a resource bundle. This string is displayed as the HTML image alt tag text. Applet strings are externalized, and the string values are passed to the applets via HTML parameters that are generated by the content transfer applet tags.

Note: When an NLS string is found, the other included files will not be processed further.

Inheriting strings

Properties files are not inherited. Specify an `nlsbundle` or `nlsclass` for your extended component. The component NLS properties file must include the properties files from the component that it extends as well as any properties files that are included in the parent component's properties file.

You can include NLS files within other NLS files. This reduces the number of NLS strings in your application and makes string values consistent across components. The included NLS files will be processed only if the NLS key is not defined in the current file. To include NLS files, add a key `NLS_INCLUDES` whose value is a comma-separated list of other property files. The following example breaks the line for display purposes, but your list should be on a single line):

```
NLS_INCLUDES=com.documentum.webcomponent.GenericActionNlsProp,com.
    documentum.webcomponent.GenericObjectNlsProp
```

NLS strings for actions are contained in the NLS resource file for the component that contains the `<dmfx:actionmenuitem>` tag. Some NLS strings for WDK actions are in `/webcomponent/strings/com/documentum/webcomponent/GenericActionsNlsProp.properties`. You can override these in the NLS properties file for the component that contains the `<dmfx:actionmenuitem>` tag. For example, the Webtop menubar component contains action menu items whose strings are located in `GenericActionsNlsProp.properties`. To override these strings, extend the menubar component and reference your own properties file. Make sure that your properties file includes `GenericActionsNlsProp.properties` so that your menus will inherit any new actions that are added to the application when you upgrade.

To add strings to an extended component:

1. Define a new resource file. For example, you are extending the renditions component and adding strings. The component definition specifies that strings are located in the resource bundle `com.documentum.webcomponent.library.renditions.RenditionsNlsProp`. This resolves to a file named `RenditionsNlsProp.properties` in the directory `/webcomponent/strings/com/documentum/webcomponent/library/renditions`. To extend this, create a file `MyRenditionsNlsProp.properties` in `/custom/strings/com/documentum/custom`.
2. Include the WDK renditions NLS resource bundle:

```
NLS_INCLUDES=
```

```
com.documentum.webcomponent.library.renditions.RenditionsNlsProp
```

3. In your extended component definition, override the strings resource with the new strings resource. For example:

```
<nlsbundle>com.documentum.custom.MyRenditionsNlsProp</nlsbundle>
```

4. Delete generated class files for JSP pages that could contain your strings.
5. Restart the application server in order to apply your changes.

Naming properties files

Every properties file in the Web application must be translated for each supported locale. NLS properties files must be named with the proper Java locale naming convention.

Properties files are named using the Java standard combination of the base name (specified in the associated resource bundle class), plus the suffix "Prop", and the code string for the supported locale. If no locale is specified in the file name, then the properties file serves as the default for applications. For a resource bundle class named `com.acme.nls.AcmeSearch`, you might have the following .properties files:

```
AcmeSearchProp_fr_CA.properties: French Canadian
AcmeSearchProp_fr.properties: French standard
AcmeSearchProp.properties: Default locale (German)
```

All of the following bundle prefixes are valid:

- [bundle name] + "Prop" + "_" + [locale language] + "_" + [locale country] + "_" + [locale variant]
- [bundle name] + "Prop" + "_" + [locale language] + "_" + [locale country]
- [bundle name] + "Prop" + "_" + [locale language]
- [bundle name] + "Prop"

Adding localized files to your application

Properties files contain the string resource data for a specific locale. They are named using the Java standard combination of the base name (specified in the associated resource bundle class), plus the suffix "Prop", and the code string for the supported locale. If no locale is specified in the file name, then the properties file serves as the default for WDK-based applications. For a resource bundle class named `com.acme.nls.AcmeNLSSearchComponent`, you might have the following .properties files:

- `AcmeNLSSearchComponentProp_fr_CA.properties`: French Canadian

- AcmeNLSSearchComponentProp_fr.properties: French standard
- AcmeNLSSearchComponentProp.properties: Default locale (German)

Overriding strings in the UI

You can override a string in the UI by hard-coding it in the JSP page. This is not recommended for strings that will be reused elsewhere in the application, but it may be necessary on occasion to change a string in one page but not in every page that it is used.

The following example overrides a string that is configured with an `nlSid` attribute in a JSP page. The original JSP page has the following control:

```
<dmf:label name="label1" nlSid="MSG_EXAMPLE");
```

Remove the `nlSid` attribute and replace it with a `label` attribute. Refer to for the specific attribute that overrides the `nlSid` for each control. In the following example, the `label` attribute overrides the `nlSid` attribute in a label control:

```
<dmf:label name="label2" label="My example");
```

Note: Delete generated class files for JSP pages that could contain your strings.

Designing for and testing internationalization

If your application will be localized, you must design it to accommodate the requirements of various locales. The following design guidelines will help you in analyzing your application:

- Externalize all strings so that they can be easily localized. Turn on the NLS strings test in the `debug_preferences` component to show strings that have not been externalized. (Refer below for details.)
- Eliminate concatenated strings. Concatenated strings assume that all languages will use the same order. Additionally, translators do not always know how the substrings are going to be put together. For example, a menu item concatenates "Undo" and "Cut" to create "Undo Cut". The concatenation in German, "Widerrufen Ausschneiden" is incorrect. You must store entire sentences in your properties files instead of sentence fragments or sentences with interpolated data.
- Design the UI for string growth. Translated strings are usually longer than the original, and they may have unpleasant effects on the UI. Turn on the long strings test in the `debug_preferences` component to show the effect of string growth. (Refer below for details.)

You can navigate to the `debug_preferences` component to turn on debugging for internationalization. You can debug the following types of errors:

- Strings that have not been internationalized
These strings will show up in the UI with an NLS key rather than a string
- Strings that will change the UI when translated to double-byte languages
- Strings that are too long for the UI

Many localized strings grow in length after translation.

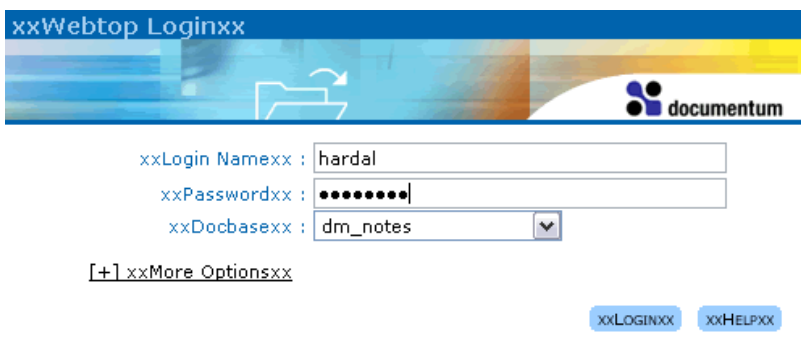
To turn on debugging for these types of errors, navigate to the preferences component and click the Debug Preferences tab using a URL similar to the following:

```
http://server_name/app_name/component/preferences
```

If a localized string does not exist, the corresponding English string is displayed. This fallback is governed by the value of the `<application>.<language>.<fallback_to_english_locale>` element in `app.xml`. Set this value to `false` for I18N testing, so that non-localized strings will be displayed as `xxNLSID_valuexx`. You can use the `NlsResourceClass` and `NlsResourceBundle` method `stringExists()` to determine whether a string exists.

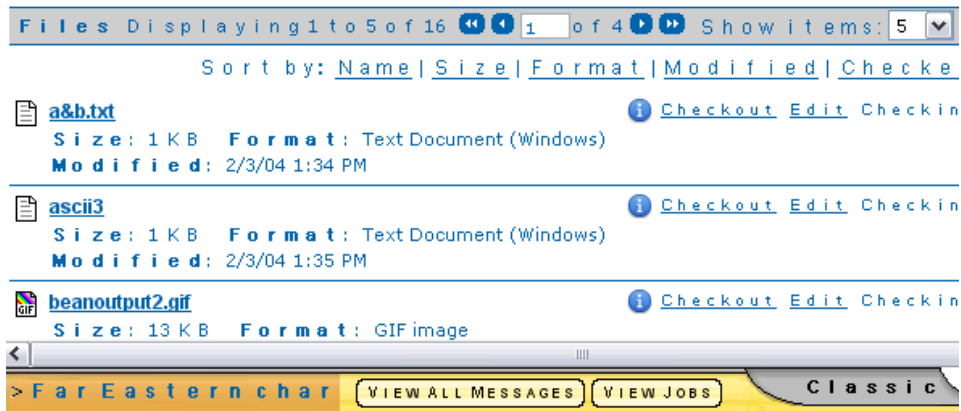
You will see "xx" surrounding each UI string for the **Test NLS Strings** checkbox. Strings that are provided by user input, queries, or image files are not affected. UI strings that you have not internationalized will not have the surrounding xx:

Figure 2-6. NLS strings test



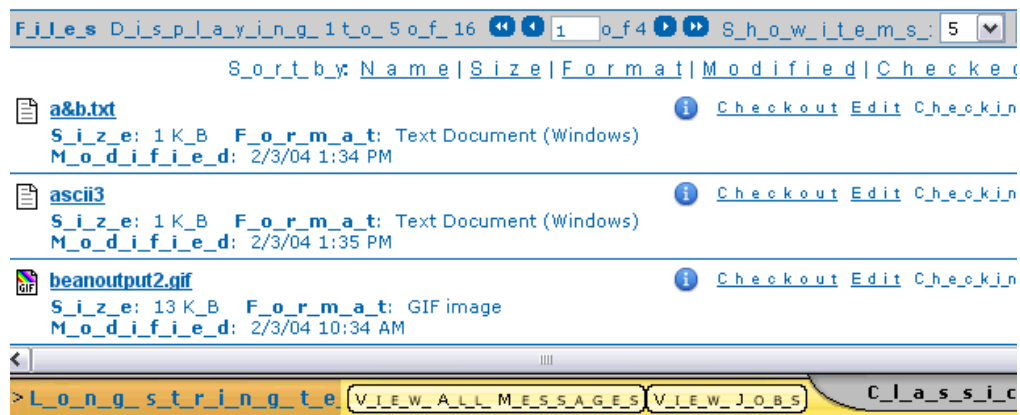
You will see each UI string displayed with a space between each letter for the **Test Far Eastern Characters** checkbox. Strings that are provided by user input, queries, or image files are not affected. Long UI strings that will negatively affect the UI when translated to double-byte languages should negatively affect the UI in this test:

Figure 2-7. Far Eastern characters test



You will see each UI string displayed with an underscore between each letter for the **Test Long Strings** checkbox. Strings that are provided by user input, queries, or image files are not affected. UI strings that will negatively affect the UI when viewed on a small screen should negatively affect the UI in this test:

Figure 2-8. Long strings test



Other tests or debugging strategies for internationalization of your application include the following:

- Check date and time display. For example, users on a German application would expect a display of 7.3.92 for the English date March 7, 1992.
- Check for truncated string inputs. Enter ASCII characters, extended ASCII characters, and double-byte characters on all text inputs to make sure they are displayed and rendered correctly.
- Check number formatting. For example, the number 123456.89 in English should be displayed as 123.456,89 in German.

- Question marks (????) or glyphs such as | or ~ instead of text in the display indicate a browser encoding problem. Make sure the browser has fonts that can display the character encoding of your application.
- Check for high ANSI characters, for example, ¼, ¶, †, %, which indicate the wrong code page for the application server.
- Delete generated class files for JSP pages that could contain your strings.

Configuring search

Search has been redesigned for the 5.3 release to enable searching across repositories and external sources. The old WDK 5.2.5 and the new WDK 5.3 search components are versioned, so that if a request is made for a search component, the new component is returned by default.

Search sources — Multiple repositories can be added to the user's search preferences. If ECI Services is installed, the user can select external sources for search and import results into the current repository. Included files within HTML or XML documents are not imported.

Search on attribute values — The attributes for search criteria are supplied by the data dictionary of the selected repository. If value assistance is defined in the data dictionary, the values are supplied for "is" and "is not" search criteria, but conditional value assistance is not implemented. Verity operators such as "not containing" or "between" are not supported.

Note: Only one repeating attribute can be queried in a single search. If repeating attributes are present in the list, they must be either of type DM_STRING or DM_ID.

Saved searches — Searches are saved as smartlist objects. Users can revise a saved search using the advanced search component. Saved searches save the display configuration as well as the query.

Smartlists created with Documentum Desktop can be executed or edited in the advanced search UI. Smartlists that are created in WDK applications cannot be used or edited in Desktop.

DQL search — The WDK 5.3 basic search component can be configured to perform a DQL search, but the DQL search is delegated to the WDK 5.2.5 search component, resulting in the search of a single repository. The user cannot set preferences for the results displayed by the DQL search.

Full-text search — Simple and advanced search support full-text queries. The search text box can contain a string within quotations marks to search for the exact string,

for example, "this string". The box also supports the operators AND and OR (not case-sensitive). Searches all indexed documents as well as indexed properties.

Value assistance — Value assistance as defined within a doc app is supported. The assistance within the doc app should provide a union of values for a type across lifecycles. The following types of assistance are supported:

- Fixed list value assistance, for example, A,B,C with no mapped labels.
- Fixed list value assistance with mapped labels, for example, A,B with labels Label A, Label B.
- Query value assistance, for example:

```
SELECT "MyDocbase"."MyTable"."MyColumn" FROM "MyDocbase"."MyTable"
```
- Conditional value assistance with optional default value assistance, for example, if condition: authors (any) = "MyAuthor".

Limitations:

- Only one repeating attribute can be queried in a single search.
- Query value assistance with a reference (`$value(attribute)`), for example:

```
SELECT "MyDocbase"."MyTable"."MyColumn1" FROM "MyDocbase"."MyTable"
WHERE "MyDocbase"."MyTable"."MyColumn2" = '$value(MyAttribute)'
```
- Not all values in value assistance may be available across repositories in a logical OR operations. (Not a limitation for AND operation.)
- Locale-based assistance values may not be available in a search of repositories on multiple locales.

The following topics describe search configuration:

- [Configuring search controls, page 145](#)
- [Configuring basic search, page 146](#)
- [Configuring advanced search, page 147](#)
- [Configuring search results, page 151](#)
- [Making search results configurable by users, page 153](#)
- [Using 5.2.5 custom search components, page 154](#)

For information on customizing search, refer to [Chapter 20, Customizing Search](#).

Configuring search controls

You can globally configure all instances of certain advanced search controls by extending the control configuration file `/wdk/config/advsearchex.xml`. The following controls can be configured:

- match case attribute on any search attribute control
- searchsizeattribute control

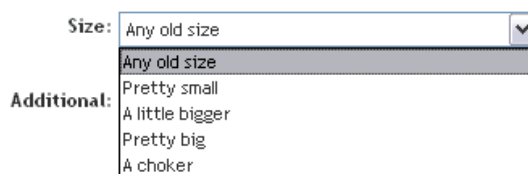
searchdateattributecontrol

The following example extends advsearchex.xml by copying it to /custom/config with the following content:

```
<config version='1.0'>
  <scope type='dm_sysobject'>
    <searchsizeattributerange>
      <option>
        <label>Any old size</label>
        <operator>LT</operator>
        <value>-1</value>
        <unit>KB</unit>
      </option>
      ...
    </searchsizeattributerange>
  </scope>
</config>
```

The resulting UI:

Figure 2-9. Search size custom dropdown list



Refer to *Web Development Kit Reference Guide* for details on each control's configuration.

Configuring basic search

Basic search searches all sysobjects in the current repository for the user-supplied string or query in indexed attributes and full-text. The default base type for the search can be configured in the search component definition. The default preferred sources can also be specified in the component definition. These sources can include multiple repositories, and external sources if ECI Services is installed.

The default search is for a string query type, which is used for a full-text search. If the content server is not configured to create a full-text index, the query is transformed into constraints against object_name, title, and subject with an OR operator. If you wish the query to include attributes, those attributes must be indexed. (All dm_sysobject attributes that are configured as searchable in the doc app are automatically indexed by the 5.3 Content Server indexer.)

If you wish to search attributes only and not full text, you must use the null query type, which is treated as DQL and passed to the 5.2.x search component. This 5.2.x search is limited to a single repository, and the user cannot configure search results columns. The results are limited to the display of dm_sysobject attributes, which are configurable in the 5.2.x search component definition.

Case sensitivity — Basic search of full-text or attributes against a 5.3 indexed repository is not case-sensitive. If the repository is not indexed, queries are case-sensitive by default. Case sensitivity for non-indexed repositories can be turned on or off in /wdk/config/advsearchex.xml, as the value of the <defaultmatchcase> element. By default it is turned off, for performance enhancement. To make case-sensitivity checkboxes display next to each property, set the casevisible attribute on the search controls to true.

Query types — The following query types can be supplied to the simple search component, which will be used to interpret the query string from the simple search box (or string supplied to the component URL):

- string
Specifies that the query string consists of one or more keywords for a full-text search
- objectId
Specifies that the query string is the object ID of a saved search
- querydef
Specifies that the string consists of the content of a smartlist
- queryId
Specifies that the string consists of the internal ID for the current query, to be used for revising the current search
- null or other value
Specifies that the string is a DQL statement and should be passed to the dqlsearchdelegate, which calls the 5.2.x search component. Search is limited to the current repository.

Note: The list of object types and their attributes comes from the reference repository. The reference repository is the first repository selected by the user. If external sources only are selected, then the list of object types in the current repository is used.

Configuring advanced search

The data dictionary provides the following data to the search UI:

- Default and other searchable attributes for a given object type
- The default and other search operators for a given type and attribute

Value assistance values for "is" and "is not" search operations, if defined

Search controls — The WDK search UI contains search controls. All search controls are contained with a `dmfxs:repositorysearch` control, which uses the values of the contained controls to build the search query. The interaction of search controls is described below. For details on the configurable attributes of these controls, refer to *Web Development Kit Reference Guide*.

- `searchlocation`

Displays the current search location. Can be hidden.
- `searchattributegroup`

Container for one or more `searchattribute` controls, which are added or removed dynamically when the user clicks the **Add** or **Remove** button. These buttons can be hidden by setting the `addvisible` attribute to `false`.
- `searchattribute`

Base control to display any single attribute. When value assistance is defined for an attribute, a dropdown list for **Equal** and **Not Equal** will be shown to allow user select an input value. If value assistance is not defined for an attribute, you can add it programmatically. (For more information, refer to [Programmatic search value assistance, page 557](#).) Subtypes of this control can be user-defined or can include the following:

 - `searchsizeattribute`

This control is configured by settings in the search control configuration file `/wdk/config/advsearchex.xml`: `<searchsizeattributerange>` See *Web Development Kit Reference Guide* for details on this configuration.
 - `searchdateattribute`

This control is configured by settings in the search control configuration file `/wdk/config/advsearchex.xml`: `<inthenextdate>`. See *Web Development Kit Reference Guide* for details on this configuration.
- `searchobjecttypedropdownlist`

Generates a list of object types from the repository data dictionary. The `basetype` attribute can filter the list for only the specified type and its subtypes. The list can be overridden programmatically by settings the options for the dropdownlist.
- `searchscopecheckbox`

Sets a value for a search setting or capability. Currently it can be used to enable finding all versions, hidden objects, or objects of a specific type.
- `searchfulltext`

Accepts keywords for a fulltext query

Case sensitivity — Advanced full-text or attributes search is not case-sensitive for a 5.3 indexed repository. The case sensitive checkboxes are not generated in the UI because the casevisible attribute is set to false on searchattribute and searchattributegroup tags.

For 5.2.5 Server or non-indexed 5.3 Server, case sensitivity is set by the value of the element <defaultmatchcase> in /wdk/advsearchex.xml. By default this value is set to true to turn on case sensitivity. Case-sensitive queries perform faster. If your environment has Server 5.2.5 or non-indexed Server 5.3 repositories only, you can add a checkbox for each attribute to the advanced search JSP page for case-sensitive search. Give the checkbox controls unique names. Set the casevisible attribute on searchattribute and searchattributegroup tags to true.

You can specify attributes for your search rather than allowing them to be generated by the searchattributegroup control. In the following example of a custom advsearch component, specific attribute controls have replaced the searchattributegroup control in the JSP page:

```

...
    <td align=left valign=top nowrap>
      <dmfxs:searchobjecttypedropdownlist name='objecttypectrl' .../>
    </td>
  </tr>

<tr><td colspan='2' class='spacer' height='10'> </td></tr>
<tr>
  <td align=right valign=top nowrap><dmf:label label='Name' cssclass="
    fieldlabel"/></td>
  <td align=left valign=top nowrap>
    <dmfxs:searchattribute name='searchname' attribute='object_name'>
    </dmfxs:searchattribute>
  </td>
</tr>
<tr>
  <td align=right valign=top nowrap><dmf:label label='Type' cssclass="
    fieldlabel"/></td>
  <td align=left valign=top nowrap>
    <dmfxs:searchattribute name='searchtype' attribute='r_object_type'>
    </dmfxs:searchattribute>
  </td>
</tr>...

```

Before this customization, the user must select properties from a dropdown:

Figure 2-10. Attribute selection dropdown

The screenshot shows the 'Advanced Search' interface with the following elements:

- General** tab selected, with sub-tabs for 'My Saved Searches' and 'All Saved Searches'.
- Contains:** An empty text input field.
- Locations:** Radio buttons for 'dm_notes; support; LisaSP1Test' (selected) and 'Current location only: dm_notes: /Lani Hardage-Vergeer'. An [Edit](#) link is present.
- Object Type:** A dropdown menu showing 'Sysobject (dm_sysobject)'.
- Properties:** A row of dropdowns: 'Name', 'contains', and an empty field. A [Remove](#) link is to the right. Below is a link: [Add another property](#).
- Date:** A dropdown menu showing 'Accessed'. Below it are radio buttons for 'Anytime' (selected), 'From', and 'To'. Each 'From' and 'To' option has a date input field and a calendar icon.

The resulting UI shows the individual attributes "Name" and "Type" as search criteria:

Figure 2-11. Specific attributes as search criteria

This screenshot shows the 'Advanced Search' interface with specific attributes selected:

- General** tab selected.
- Contains:** An empty text input field.
- Locations:** Radio buttons for 'dm_notes; support; LisaSP1Test' (selected) and 'Current location only: dm_notes: /Lani Hardage-Vergeer'. An [Edit](#) link is present.
- Object Type:** A dropdown menu showing 'Sysobject (dm_sysobject)'.
- Name:** A dropdown menu showing 'Name', a dropdown menu showing 'contains', and an empty text input field. A [Remove](#) link is to the right.
- Type:** A dropdown menu showing 'and', a dropdown menu showing 'contains', and an empty text input field. A [Remove](#) link is to the right.
- Date:** A dropdown menu showing 'Accessed'. Below it are radio buttons for 'Anytime' (selected), 'From', and 'To'. Each 'From' and 'To' option has a date input field and a calendar icon.

If you wish to display specific custom attributes as search criteria, extend the advanced search component, scope the definition to your custom type, and provide a custom JSP page. In that page, add attribute controls for your attributes. When the user selects the custom type, the scoped definition will be read by the configuration service and the custom JSP page will be displayed. The custom attributes will be displayed similar to the following:

Figure 2-12. Custom attributes as search criteria

The screenshot shows a search configuration interface with the following elements:

- Tabs: General, My Saved Searches (selected), All Saved Searches
- Contains:
- Locations:
 - dm_notes; support [Edit](#)
 - Current location only: dm_notes: /Lani Hardage-Vergeer
- Object Type: Web Document (technical_publications_web) ▼
- Criteria List:

Name	contains ▼	<input type="text"/>	Remove
Type	and ▼	contains ▼	<input type="text"/> Remove
Edition	and ▼	contains ▼	<input type="text"/> Remove
OK to Display?	and ▼	= ▼	true ▼ Remove

Configuring search results

You can configure the maximum number of search results and turn off term hit highlighting. After you have made custom types and their attributes available for search, you can configure the display of custom attributes in the search results. You can configure the `display_preferences` component to allow users to configure their preferences for displaying custom attributes.

Maximum number of search results — The maximum number of search results is configured in `dfc.properties`. The maximum number of search results is divided by the number of search sources. For example, if a maximum of 1000 results is specified in `dfc.properties`, and the user specifies 10 search sources, only 100 results will be displayed for each source. If some sources return fewer than 100 results, the other sources do not display more than 100 results.

Term hit highlighting — Term hit highlighting (highlighting of the search term in the results) can be set as a user preference. The default value is set as the value of the element `<highlight_matching_terms>` in the search component definition, which is located in

/webcomponent/config/library/search/searchex. If you are customizing Webtop or an application that extends Webtop, you must add a <highlight_matching_terms> element to the top-level search component definition.

Configuring the display of attributes in search results — Default search result columns are configured as <column> elements in the basic search configuration file search_component.xml in /webcomponent/config/library/search/searchex. Because Webtop extends this definition, your customization in /custom/config should extend the Webtop definition and override the <columns_drilldown>, <columns_list>, and <columns_saved_search> elements in your definition. Only searchable attributes can be specified as columns. Users can set a preference for search results columns in the display_preferences U, which will then override the default settings in the configuration file.

Your custom search component definition must specify a scope for the custom type in order to define visible columns for custom attributes. For example, if the user selects a custom type for the advanced search, the columns specified in your scoped basic search component will be displayed in the results. Details of the columns configuration can be found in *Web Development Kit Reference Guide*

In the following simple configuration, the definition extends the WDK search component definition and adds some custom attribute columns:

```
<scope type='technical_publications_web'>
<component id="search" extends="
  search:webcomponent/config/library/search/searchex/search_component.xml">

<nlsbundle>com.documentum.webcomponent.library.search.SearchExNlsProp</nlsbundle>
<type>technical_publications_web</type>

<columns_drilldown>
  <loadinvisibleattribute>true</loadinvisibleattribute>
  <column>
    <attribute>object_name</attribute>
    <label><nlsid>MSG_NAME</nlsid></label>
    <visible>true</visible>
  </column>

  <column>
    <attribute>tp_edition</attribute>
    <label>Edition</label>
    <visible>true</visible>
  </column>

  <column>
    <attribute>tp_web_viewable</attribute>
    <label>OK to display</label>
    <visible>true</visible>
  </column>
  ...

```

The following example shows the results of a scoped basic search component with custom attributes:

Figure 2-13. Custom search results for custom type

The screenshot shows a search results page titled "Search Results" for the query "WDK" in the "dm_notes" index. The page includes navigation controls like "Page 1 of 2+", "Items per page: 10", and sorting options: "Name", "Edition", "OK to display", "Format", and "Modified".

Document Name	Edition	OK to display	Format	Modified	Actions
WDK_Install_Rel_Notes_423_WIN_SOL.pdf	4i	1	Acrobat PDF	12/16/02 12:25 PM	Checkout Edit Checkin Cancel Checkout Add To Clipboard More...
TroubleshootWDK.pdf	Not Applicable	1	Acrobat PDF	12/16/02 12:14 PM	Checkout Edit Checkin Cancel Checkout Add To Clipboard More...
Using_WDK_51.pdf	Documentum 5	1	Acrobat PDF	3/10/03 2:37 PM	Checkout Edit Checkin Cancel Checkout Add To Clipboard More...
WDK.chm	Not Applicable	1	Unknown	12/16/01 1:53 PM	Checkout Edit Checkin Cancel Checkout Add To Clipboard More...

Making search results configurable by users

The user can select attributes for display in search results, which overrides the default display. The preferences UI allows users to specify the attributes that will be displayed for specific object types. If the user configures different display columns, the query will not be reissued, so data may not be displayed in the new columns until the search is performed again. Calculated columns such as score or summary will not display any values unless they are selected before the query is run.

To make a custom type available in preferences

1. Extend the `display_preferences` component in your `custom/config` directory. Change the `<component>` element as follows:

```
<component id="display_preferences" extends="
  display_preferences:webcomponent/config/environment/preferences/
  display/display_preferences_component.xml">
```

2. Add your custom type to the `<display_docbase_types>` element. For example:

```
<display_docbase_types>
  <docbase_type>
    <value>dm_document</value>
    <label><nlsid>LBL_DOCUMENT</nlsid></label>
  </docbase_type>
  ...
  <docbase_type>
    <value>my_custom_type</value>
```

```
<label>My type</label>
</docbase_type>
</display_docbase_types>
```

Note: You can also put a `<display_docbase_types>` element within a `<preference>` element in this component. This will make your custom type available for display only in the component that named within that preference. (The component is named within the `<preference>.<value>` element.)

3. Save this file and refresh the configuration files on the application server by navigating to `/wdk/refresh.jsp`.

Using 5.2.5 custom search components

The old (5.2.5) and new search components are versioned, so that if a request is made for a search component, the new component is returned by default. Customizations that extend 5.2.5 search components are supported and do not require versioning. That is, if your custom search component extends the Webtop search or advsearch component in `search_component.xml` or `advsearch_component.xml`, it should work out of the box. If you are not extending Webtop, your search component can extend the `dqlsearchdelegate` component, which simply extends the 5.2.5 search component.

To take advantage of new search functionality, you can extend the new search components by extending `search` or `advsearch` in `searchex_component.xml` or `advsearchex_component.xml`.

To use your customized WDK 5.2.5 search component, make sure your component and container extend the WDK 5.2.5 component and container.

A special component, `dqlsavesearchdelegate`, executes the view action for saved 5.2.5 queries by extending the 5.2.5 `savesearch` component.

Packaging and deploying Web applications

The contents of a Web application must conform to the J2EE directory structure for Web applications as specified in the J2EE Servlet specification. WDK supplies the required content for a Web application. You must include your application content within an installed WDK application such as WDK, Webtop, or Web Publisher.

Documentum Web applications are packaged into a Web application archive (WAR) for deployment on a J2EE-compliant application server. The following steps will ensure a deployment to a production application server from your development environment.

Deploying to a production application server

1. Customize a WDK-based application such as Webtop on a certified environment. (Refer to the release notes for the certified application server environments.)
2. If your development application server is not Tomcat, copy your customizations over to an installation of the WDK-based application on a certified Tomcat environment.
3. Package the WAR using a Java tool such as the tool provided by WDK (refer to [WAR packaging tool, page 155](#)).
4. Use the WDK-based application installer for your production operating system to deploy the application. (The installer will add Documentum native libraries to the J2EE server host machine and configure a start script that starts the Web application with the appropriate classpath and library references.)



Caution: The installer must be correct for the application server operating system. For example, you cannot use a Windows installer to deploy onto a UNIX operating system.

The following topics describe Web application packaging:

- [WAR packaging tool, page 155](#)
- [Deploying with the application installer, page 156](#)
- [Development update tool, page 156](#)
- [Compiling and precompiling JSP pages, page 157](#)

WAR packaging tool

You can use the tool CreateInstallerWAR to package your Web application for deployment. You must use this tool on a certified Tomcat application server environment.



Caution: You cannot deploy the packaged WAR file within an application server. You must place the packaged WAR file within the WDK or WDK-client installer, because the installer adds Java libraries, native libraries, and classpath information to the application server start script. Refer to [Deploying with the application installer, page 156](#) for more information.

The CreateInstallerWAR tool strips out all comments from files with the following extensions: .html, .htm, .js, and .jsp. This improves the performance of your Web application.

To create a WDK-based WAR file

1. Ensure that you have the Sun Java SDK root directory on your system path.

2. Include \WEB-INF\classes on your classpath:

```
set
  classpath App_root_directory\WEB-INF\classes
```

where *App_root_directory* is the root directory for your custom Web application.

3. From the command prompt, enter the following command on a single line:

```
java com.documentum.web.tools.CreateInstallerWAR
  source_virtual_directory
  destination_file
```

where *source_virtual_directory* represents the Tomcat root directory for your custom Web application and *destination_file* represents a name for your new WAR file.

4. Remove the entry in your system classpath that you created in step 2, in order to run the WDK application.

Deploying with the application installer

You must use the WDK or WDK client application installer to deploy your packaged WAR file. For example, if you have customized Webtop, use the Webtop installer to deploy your custom WAR file. The installer adds Java libraries, native libraries, and an appropriate start script for the application server.



Caution: The installer must be correct for the application server operating system. For example, you cannot use a Windows installer to deploy onto a UNIX operating system.

To package a custom WAR file within the installer:

1. Navigate to the directory that contains the WDK or Webtop installer and expand the installer archive file.
2. Name your WAR file *wdk.war* if you are using the WDK installer for a development update. Name your WAR file *webtop.war* if you are using the Webtop installer for deployment or *wp.war* if you are using the Web Publisher installer.
3. Copy the new WAR file to the installer directory, replacing the war file in that directory.
4. Run the installer. It will deploy the new WAR file.

Development update tool

WDK contains a deployment tool `ExpandInstallerWAR` that expands an archive (`unstripped.jar` in the WDK installer) to a development directory. You can use this tool to update Web applications that are under development.

If the destination directory already contains WDK files, and the file size of an existing file is different from the file size of a file in `unstripped.jar`, the existing file is renamed with an extension ".old" before the file in `unstripped.jar` is expanded. This protects your customized files from being overwritten.

Note: If you follow Documentum's recommended customization model, you do not modify any of the files in the WDK installation. All of your custom files are contained within the `/custom` directory or other new directory of your choice, and your custom class files are contained within their own directory under `/WEB-INF`. The `ExpandInstallerWAR` tool will not overwrite your custom files.

To install the WDK JSP pages with comments:

1. Ensure that you have the Sun Java SDK on your system path.
2. Ensure that you have run the WDK installer, that you know the location of the WDK-based WAR file, and that a target directory exists for the expanded Web application.
3. Include the path to `WEB-INF/classes` on your classpath. For example (Windows):

```
java -cp Web_root_directory\WEB-INF\classes
```

4. At the command prompt, enter the following command:

```
java com.documentum.web.tools.ExpandInstallerWAR
    source_war destination_virtual_directory
```

where *source_war* represents the root directory for your custom Web application and *destination_virtual_directory* represents the name of the directory that will contain your expanded Web application.

Compiling and precompiling JSP pages

The first time a user loads a particular JSP into the browser, the JSP file and any graphics and other JSP pages or files that it includes are first translated into a Java source file (.java) and then compiled into a .class file. When the user makes a request of the application (for example, by entering data into a form and clicking **Submit**), the application handles any data the user submits or retrieves data dynamically from the repository and returns the data to the .java file, where it is recompiled in the .class file. The .class file returns the data to the client Web browser.



Caution: (WebSphere) The `java.compiler` option must be set to `none` for compilation of WDK JSP pages to work properly. This value is set by the WDK installer. You can verify this setting in the administration client console by expanding the WDK application node and application server. The JVM settings tab displays System properties including `java.compiler` and `java.library.path`, both of which are modified by the installer.

Most J2EE application servers will compile a JSP page into a Java class the first time the page is requested. To enhance performance, precompile the application JSP pages before starting the application.

If your application server does not have a compiler, you must compile the application JSP pages using an external compiler. You must set the classpath for the precompiler to include all of the contents of the classpath as configured by the WDK installer in the J2EE server start script. That is, when you run the WDK installer, it creates a start script for the application server that contains the required classpath.

WDK and WDK-based applications do not contain precompiled JSP classes, because each application server has its own naming convention and mapping between JSP pages and the corresponding class.

Configuring Controls

A control is a Java object that models the attributes of HTML UI elements. The application user interface (UI) is built from controls that generate HTML and maintain control state on the server. The user interface design is configured by setting JSP tag library attributes for a control tag class. The control state is maintained on the server by the control class.

Each control has a corresponding control tag class that initializes the control and generates the user interface. The tag class implements tag library accessor methods to get and set the control's attributes. You can configure controls by setting these attributes on the JSP page. The control tag class then generates HTML and JavaScript elements that are rendered into an HTML page to the browser.

Some controls fire events that are handled either on the client or the server. The component class that uses the control in a component JSP page defines an event handler method to handle the server-side control events.

In a desktop-style application design, the user selects operations and views from menus. Menu controls generate cascading menus for classic or list-style applications. In a Web-style application design, the user selects sets of operations or views through links of buttons on tabs. Tab controls generate tabs for streamline or drilldown applications.

The following topics describe controls:

- [What controls do, page 160](#)
- [How to configure controls, page 160](#)
- [Finding files to configure controls, page 162](#)
- [Using tag libraries, page 164](#)
- [Control events, page 165](#)
- [Types of controls, page 169](#)
- [Action-enabled controls, page 171](#)
- [Controls that can be globally configured, page 175](#)
- [Hiding controls, page 177](#)
- [Configuring dates, page 177](#)
- [Configuring menus, page 178](#)

- [Configuring tabs](#), page 181
- [Configuring dropdown lists](#), page 181
- [Configuring scrollable controls](#), page 183
- [Configuring databound controls](#), page 184
- [JSP fragment control](#), page 189
- [Configuring rich text](#), page 190
- [Displaying and validating attributes](#), page 192
- [Validating user input](#), page 202
- [Working with images and icons](#), page 206
- [Working with tooltips](#), page 208

For information on individual controls and their configuration, refer to *Web Development Kit Reference Guide*.

What controls do

You can use controls for the following purposes:

- Accept user input (refer to [Types of controls](#), page 169)
- Raise events that change the behavior of the component that contains the control (refer to [Control events](#), page 165)
- Format output (refer to [Types of controls](#), page 169)
- Change the display of repository data (refer to [Types of controls](#), page 169)
- Launch an action (refer to [Action-enabled controls](#), page 171)
- Bind to and display data (refer to [Configuring databound controls](#), page 184)
- Display a different set of attributes for each component UI (refer to [Displaying and validating attributes](#), page 192)
- Validate user choice (refer to [Validating user input](#), page 202)

How to configure controls

Each Documentum control has a corresponding control tag in the WDK tag libraries. Controls are configured by setting attributes on the control tag in a JSP page.

Tags are organized by functionality into the following libraries in the folder /WEB-INF/tlds:

- `dmform_1_0.tld`

Contains tags that generate basic controls, such as buttons, links, lists, and trees. These controls support data binding to generic data sources (JDBC) as well as DFC connections.

- dmformext_1_0.tld

Contains repository-enabled controls that can display values from the repository or validate user input based on the repository data dictionary.

- dmwebtop_1_0.tld

Contains Webtop-specific controls. (Installed by the Webtop installer.)

- dmcontentxfer_1_0.tld

Contains tags that generate content transfer applets.

- dmda_1_0.tld

Contains controls used for repository administration.

- dmfxsearch_1_0.tld

Contains controls that are used by the advanced search component (advsearch)

- dmlayout_1_0.tld

Contains controls that generate HTML layout tags



Caution: You must use action controls or controls that get data from a repository (dmfx:...) within a component. Documentum control tags will not work properly in JSP pages that are not within the component framework.

Follow this procedure to configure JSP control tags when you configure Webtop or a WDK-based application:

To configure a control tag:

1. Name the control. The control name is an attribute of the control tag. Named controls are cached on the server and maintain state when the user navigates through browser history. Controls with the same name are indexed automatically.
2. Set the control tag attributes. Most controls have common attributes that you can set by assigning values to JSP attributes, such as *cssclass*, *datafield*, *enabled*, *name*, *nlsid*, *onchange*, *runatclient*, *style*, and *visible*. Individual controls can also have attributes specific to the control. For specific control attributes, refer to *Web Development Kit Reference Guide*.
3. Add any arguments. Some controls take arguments that are passed with control events.
4. Specify the control event handlers. For server-side event handlers, the event handler method is named in an event attribute. For client-side events, set the *runatclient* attribute to true and add a JavaScript event handler on the JSP page. (Dynamic action control events cannot be handled on the client.)

Example 3-1. Adding Control Tags

Add control tags into the JSP within the HTML elements (<html> and </html>) of the page. The following example adds a help button in a table cell (<td>):

```
<td>
  <dmf:button name='Help' cssclass="buttonLink" nlsid='MSG_HELP'
    onclick='onClickHelp' runatclient='true' height='16'
    imagefolder='images/dialogbutton' />
</td>
```

In the above example, the button tag is configured with the following settings:

- **cssclass:** Specifies a class that sets the style for the button.
- **nlsid:** Specifies a lookup key that will be replaced by a localized string at run time.
- **onclick:** Specifies the name of the function that will be called when the button is clicked.
- **runatclient:** Specifies that the event will be handled by a JavaScript event handler on the client, not the server.
- **height:** Specifies the image height, which will be rendered as an HTML attribute.
- **imagefolder:** Specifies the location of the control images. An absolute folder path (leading '/') is relative to the Web virtual directory. A relative path (no leading '/') is relative to a theme folder.

Some controls have additional configuration through an XML configuration file. For information on these controls, refer to [Controls that can be globally configured, page 175](#).

Finding files to configure controls

The files that control a particular feature in the UI or a particular application behavior may be located in more than one application layer and in more than one directory within an application layer. (Application layers are described in [Application layer inheritance, page 42](#).) The following instructions describe how to find the right files in order to configure a control:

1. Locate the string in the UI that accesses the feature you wish to configure
2. Find this same string in a properties file in the highest application layer.
3. Find the XML configuration file that uses this properties file
4. Copy the XML configuration file and JSP page into your custom directory for configuration.
5. Make your changes in your custom JSP page.
6. Refresh the application server configuration cache and view your changes.

To find configuration files for buttons or links:

1. Identify a string in the UI that is associated with the feature of interest. For example, in the Webtop streamline view of the Home Cabinet tab, you see a series of buttons across the top. You are interested in launching your own custom search component from the **Advanced** button.
2. Using a multi-file search tool, which most IDEs have, search all *.properties files for the string in the /strings directory of the top folder in your Web application. For example, you would first search /custom/strings to see whether your application has already customized this button. Then you would search /webtop/strings. If you don't find it in /webtop/strings, search /webcomponent/strings and last, /wdk/strings.
In this example, the string "Advanced..." is found in the file TitleBarNlsProp.properties. The key to this string is MSG_ADVANCED_SEARCH, which you will find later in the JSP page that you customize.
3. Search *.xml files for the configuration file that contains this resource bundle. Drop the .properties, because the bundle name does not have an extension. In this example, you search on "TitleBarNlsProp.". Use the same application layer hierarchy to search that you used to search for the string.
TitleBarNlsProp is found in /webtop/config/titlebar_component.xml. This is the component you will need to extend, because the advanced search button on the JSP page must launch your custom component.
4. Copy the configuration file to your /custom/config directory and open it in your editor.
5. Locate the <pages>.<start> element, which contains the reference to the start page /webtop/titlebar/titlebar.jsp. Change this to /custom/titlebar.jsp and save and close the XML file.
6. Copy the original component JSP page to your custom directory so that you can edit it. In this example, copy /webtop/titlebar/titlebar.jsp to /custom and open the file in your editor.
7. Locate the UI feature that will launch your custom component. In this example, search on the NLSID key MSG_ADVANCED_SEARCH. You will find this key as the nlsid attribute value for a button tag. Note that the onclick attribute specifies an event handler, onclick="onClickAdvancedSearch". The runatclient attribute is true, so you will search for this event handler on the JSP page. In our example, the event handler is on the page. If you don't find it on the page, another JSP page may have registered this event handler, so you would have to search for onClickAdvancedSearch in your JSP pages using the same application layer hierarchical search that you used previously.
8. In the event handler, find the call to nest to the advanced search component:

```
function onClickAdvancedSearch()  
{
```

```

var contentPage = eval(getAbsoluteFramePath("content"));
if (contentPage != null)
{
    postComponentNestEvent( null,
        "advsearchcontainer", "content", "component", "advsearch");
}
}

```

Change this call to nest to your custom component:

```

{
    postComponentNestEvent( null,
        "customadvsearchcontainer", "content", "component", "customadvsearch");
}

```

(You will need a custom search container that specifies your custom search component within it, because the default search container contains the default advanced search component).

9. Save and close your JSP page.
10. Refresh the configuration files in memory by navigating to `/wdk/refresh`, and then view the page that contains your customized button or link. Your custom component should be launched when you click the button or link.

Using tag libraries

WDK provides JSP tag libraries of custom tags. The tag libraries must be included in your Web application, and you can add your own tag libraries. The Documentum tag libraries are located in the `/WEB-INF/tlds` directory.

The following table lists the WDK tag libraries and their purposes:

Table 3-1. WDK tag libraries

Library	Prefix	Purpose
Basic dmform_1_0.tld	dmf	Generates basic HTML and JavaScript
Extended dmformext_1_0.tld	dmfx	Generates HTML and JavaScripts using repository data
Content transfer dmcontentxfer_1_0.tld	dmxfer	Generates content transfer applets
Search dmfxsearch_1_0.tld	dmfxs	Generates search controls

Library	Prefix	Purpose
Layout dmlayout_1_0.tld	dml	Not used
Administrator dmda_1_0.tld	dmda	Generates administrator HTML and JavaScript

Tags in the WDK tag library are referenced with prefixes such as dmf, dmfx, or dmxf. To include the tag library in a JSP page, insert the directive at the beginning of the page. For example:

```
<%@ taglib uri="/WEB-INF/tlds/dmform_1_0.tld" prefix="dmf" %>
```

To insert a tag from the tag library, use the tag name with the prefix. For example, the version label is inserted as follows:

```
<dmf:label name="version" nlsid="MSG_VERSION" cssclass="checkinFontVerySmall"/>
```

Control events

A WDK control such as a button or text input field can fire one or more events or it can launch an action. Most control events are handled on the server by an event handler that is named as the value of the event attribute. For example, the following button control specifies an onclick event handler that corresponds to a handler method onClickMakeCurrent():

```
<dmf:checkbox name="makecurrent" value="true" nlsid="MSG_MAKE_CURRENT"
  onclick="onClickMakeCurrent"/>
```

Some control events submit the form and are handled immediately. Other control events do not submit the form and are handled only after the form has been submitted. For example, the controls that have an onclick or onselect event submit the form. Controls that have an onchange event do not submit the form. Controls that have an onload and onunload event are handled when the browser loads or unloads the page. The onfileselect event renders a postServerEvent function call, which is handled immediately.

The server-side event handler must be in the class of the component that owns the control's JSP page, in a parent class of the component, or in the class of the container that contains the component. In the following example, when the user clicks the **Show All** checkbox in the attributes page attributes_all.jsp, the onclick event specifies an onShowAllClicked event handler, as follows:

```
<dmf:checkbox name='show_all' onclick='onShowAllClicked'
  nlsid='MSG_SHOW_ALL_PROPERTIES'/>
```

This handler is in the component class Attributes:

```
public void onShowAllClicked(Checkbox showAll, ArgumentList arg)
{
    if ( showAll.getValue() )
```

```
{
    // show the layout for all properties
    setComponentPage("all");
}
...
}
```

Control events are described in the following topics:

- [Types of control events, page 166](#)
- [Configuring control events, page 168](#)
- [Control event arguments, page 168](#)
- [Handling a control event on the client, page 168](#)

For information on general client-side pre-processing before a JSP form is submitted, refer to [Presubmission client events, page 242](#).

Types of control events

Controls can fire the following types of events:

- Control state change events

Control state change events are raised on the server when the state of a control has changed. All accumulated state change events are invoked after the form is submitted. You cannot configure these events unless the event is set to run on the client (the control attribute `runatclient = "true"`). Dynamic action control events cannot be handled on the client.

- User-originated events (refer to [Control events, page 165](#))

User-originated events are raised by controls in reaction to user actions in the UI, such as clicking a **Close** button. Includes action events, where the control supports an action attribute that calls an action. (Refer to [Action-enabled controls, page 171](#) for more information on control action events.)

Many controls that accept user input support the following event attributes. The value of the event attribute corresponds to the name of an event handler. The event is generally handled in the calling component class unless the control has the `runatclient` attribute set to true. To find the exact event attributes that are supported by a control, refer to the tag library descriptor (*.tld file) for that control.

Table 3-2. Event attributes

onchange	Sets the event that is fired when the control is changed by the user. The onchange event handler cannot run on the client (runatclient cannot be true). The onchange event is not handled immediately. It is handled on the server only when the form is submitted (for example, by an onclick event). Some controls do not implement an onchange event.
onselect	Sets the event that is fired when the user selects the control, such as an option in a list. The onselect event is handled immediately, either on the server (when runatclient="false") or on the client (when runatclient="true"). Some controls do not implement an onselect event.
runatclient	Specifies that the onchange event should run on the client, not the server. Some controls do not support a runatclient attribute.
onclick	Sets the event that is fired when the user clicks the control, such a button. The onclick event is handled immediately, either on the server (when runatclient="false") or on the client (when runatclient="true"). Some controls do not implement an onclick event.
defaultonenter	If set to true, calls the keyboard Enter key JavaScript event. Another control on the page must have a default attribute set to true, so that when a user clicks the Enter key, the default control event is fired. For example, the JSP page aclist.jsp contains a text tag with defaultonenter="true". The Jump To button has the attribute default="true." When the user enters a value in the text box, the Jump To button uses that value to navigate. It is the responsibility of the default control event handler to get the value of the defaultonenter control.

Refer to the individual control descriptions in *Web Development Kit Reference Guide* for the specific events that are defined for each control.

Configuring control events

Controls that fire events have one or more event attributes that you can set on the JSP. You can specify a client-side event handler for a control event by setting the `runatclient` attribute value to `true` if the `runatclient` attribute is specified in the tag library for the control. Additionally, WDK provides JavaScript resource files to support modal windows, navigation, progress bar, scrolling, tracing, and other client functionality.

Control events are often handled in a component class. This allows a control to be used in different ways depending on its component context. For example, an OK button can be handled one way in the import component and a different way in a delete component. For information on configuring control events, refer to [Control events, page 165](#)

Control event arguments

You can add custom arguments using the `<argument>` tag to an event handler using the `<argument>` tag. WDK does not limit the number of arguments that can be specified. For example:

```
<script>
  function handleClick(srcObject, customArg1)
  {
    alert("Argument = " + customArg1);
  }
</script>
...
<dmf:button name='mybutton' onclick='handleClick'
  runatclient='true'>
  <dmf:argument name='anArgument' value='One' />
</dmf:button>
```

Custom `<argument>` tag values are passed as JavaScript function arguments in the order in which the arguments are defined in the JSP. For the event handler to gain access to the custom arguments, the function signature must include the initial object argument, regardless of whether it is used or not, plus one argument for each `<argument>` tag. The names of the JavaScript function arguments are arbitrary and do not have to match the names specified in the `<argument>` tags.

Handling a control event on the client

Sometimes events must be handled within the browser, for example, to implement dynamic HTML behavior. To handle the event on the client, set the control attribute `runatclient` to `true`. (Consult the tag library to make sure the `runatclient` attribute is supported for the control.) This forces all events fired by the control to be handled on the client by a registered event handler.

To handle a client-side event, add the JavaScript event handler function to the same layout JSP page that contains the event-firing control, or include a JavaScript file that contains the event handler. The event handler function name must match the control's event handler name. In the following example, the button specifies a handleClick event handler for the onclick event. Because the runatclient attribute is set to true, the event handler must be on the client:

```
<script> function handleClick()
{
    alert("Button click has been handled!");
}
</script>
<dm:button name="mybutton"
    onclick='handleClick' runatclient='true'>
</dm:button>
```

Types of controls

Basic controls in WDK generate HTML and JavaScript that is sent to the browser. These controls are located in the com.documentum.web.form package and subpackages. Additional controls that get or set data in the repository are provided in the com.documentum.web.formext package and subpackages.

The following types of controls are provided in the WDK tag libraries:

- Basic

The basic controls in the com.documentum.web.form.control package generate HTML widgets. These tags are defined in the tag library dmform_1_0.tld.

- Repository-enabled

The repository-enabled controls in the com.documentum.web.formext.control package get information from the repository using the current session. Many of them perform attribute validation. These tags are defined in the tag library dmformext_1_0.tld.

- Action-enabled

Action-enabled controls in the package com.documentum.web.formext.control.action launch an action that is specified as the value of the action attribute. Refer to *Web Development Kit Reference Guide* for more information on this attribute. These tags are defined in the tag library dmformext_1_0.tld.

- Content transfer

The content transfer controls in the package com.documentum.web.contentxfer.control and their configuration attributes are used by WDK 5.2.5 applet content transfer. Refer to *Web Development Kit Reference Guide* for more information. These tags are defined in the tag library dmcontentxfer_1_0.tld.

For information on configuring a specific control, refer to *Web Development Kit Reference Guide*.

Non-input controls — Several controls do not accept user input. They generate output in the form of HTML and JavaScript to the browser. Each non-input control has configurable attributes that affect the display and events of the control. Examples of non-input controls include `browserrequirements`, `button`, `formurl`, `image`, `label`, `link`, `menu`, `menuitem`, `menuseparator`, `menugroup`, `option`, `panel`, `columnpanel`.

Boolean input controls — Two tags (`radio` and `checkbox`) support true or false selection by the user.

String input controls — String input controls accept user text input. Examples include `breadcrumb`, `dateinput`, `datetimeinput`, `dropdownlist`, `listbox`, `hidden`, `password`, `row`, `tab`, `tabbar`, `text`, `textarea`, `tree`.

Format controls — Format controls are read-only controls that format the contained control's value into another form. They are particularly useful for formatting data values within datagrids. Examples of format controls include `docformatvalueformatter`, `docsizvalueformatter`, `folderexclusionformatter`, `rankvalueformatter`, and `vdmbindingruleformatter`.

Databound controls — Many WDK controls including `label`, `text`, `button`, `link`, `panel`, and `hidden` can bind to data as a source for one of the control attributes. Databound controls implement `IDataboundControl` or extend a class that implements it, for example, `datadropdownlist`, `datalistbox`, and `datagrid`. The `datafield` attribute on a databound control specifies the field that provides the data for the control attribute. When the `datafield` attribute is set, the control is said to be databound.

Note: Any component that uses a databound control in a JSP page must establish a session and data provider for the control. Databound controls cannot be used on JSP page that doesn't have a Documentum session. Refer to [Providing data to databound controls, page 186](#) for information on establishing a session and data provider.

Specialized controls aid in the data binding process, for example, providing data, supporting sorting and paging. Data binding support controls in the package `com.documentum.web.form.databound` include `cellist`, `celltemplate`, `datadropdownlist`, `datapagesize`, `datalistbox`, `dataoptionlist`, `datasortlink`, `datagrid`, `datagridrow`, and `nodatarow`.

Component Controls — WDK contains controls that assist components. Component controls include `componentinclude`, `componenturl`, `containerinclude`.

Media Controls — WDK contains one media control: `thumbnail`. This is a media server thumbnail control that is integrated with Content Server to display thumbnail images.

Attribute controls — Attribute controls leverage data dictionary information by displaying icons, labels, or values based on the Documentum object associated with the control. For example, the rename component JSP page (rename.jsp) includes the `docbaseobject` control, then gets and displays the label and value:

```
<dmfx:docbaseobject name='obj' />
...
<dmfx:docbaseattributevalue
  object='obj' attribute='object_name' />
<dmfx:docbaseattributevalue object='obj'
  attribute='r_lock_owner' />
```

The `docbaseattribute` tag renders a label and attribute value. The attribute can be displayed as read-only or editable. The type of HTML widget that is rendered to display the attribute is based on the attribute datatype (for example, checkbox, date/time, textbox). If the attribute is an editable date, a `dateinput` tag is rendered, and the date range for this control is configured in the `/wdk/app.xml` file.

Attributes can be displayed in attribute lists based on context such as type, role, or current component. The lists are configured either in the data dictionary or in a list configuration file.

For general information on configuring attributes and attribute lists, refer to [Displaying and validating attributes, page 192](#).

Action-enabled controls

Action-enabled controls are automatically hidden or disabled if the associated action is not resolved or one of the preconditions is not met. Buttons, menu items, links, or checkboxes can be action-enabled. For example, if a user does not have permissions to delete a document, the delete option may be grayed out or hidden.

Specify the action associated with the action control as the value of the `action` attribute in the JSP control tag. The action is then matched by the action service to an action definition. Optional nested argument tags pass additional arguments from the context or query resultset to the action.

Action controls can launch a single action, such as an action button, link, or menu item. Simple action controls specify the action as the value of the `action` attribute. Action controls can also launch more than one action, such as an action button list and action link list. The actions in the list are defined in an action XML definition.

For information on implementing multiple selection, refer to [Implementing multiple selection, page 415](#).

The following topics described action controls:

- [Types of action controls, page 172](#)

- [Dynamic action controls, page 172](#)
- [Using dynamic action controls, page 174](#)
- [Controlling visibility, page 174](#)

Types of action controls

Controls are defined as static when they are configured by an action attribute in the JSP page. Action execution is performed when the control action button, link, menu item, or multiple checkboxes are selected. These controls do not support the "dynamic" attribute, which is described in [Dynamic action controls, page 172](#).

Dynamic action controls

The visibility and enabled state of a dynamic action control are based on the run time state of other action controls on the same JSP page. This dynamic behavior is configured using the dynamic attribute of an action control. Dynamic action controls do not support the `runatclient` attribute.

multiselect — The `multiselect` attribute specifies whether an action can be performed on multiple objects. If a control's dynamic attribute is set to `multiselect`, the control's associated action can be invoked on one or more selected objects, if the action preconditions are met. For example, in `permissions.jsp` the `Edit` `actionimage` control has the argument `dynamic="multiselect"`. The user can launch the `edit` action for multiple selected objects.

Refer to [Implementing multiple selection, page 415](#) for more information on using this attribute. This `multiselect` attribute should not be confused with the `actionmultiselect` control tag.

singleselect — If a control's dynamic attribute is set to `singleselect`, the action can be invoked when one and only one object is selected, if the action preconditions are met. For example, the `Webtop toolbar.jsp` page contains a **properties** button that is enabled only when one object checkbox is selected:

```
<dmfx:actionbutton dynamic='singleselect' showifdisabled='true'
  showifinvalid='true' name='properties' action='properties'
  nlsid='MSG_PROPERTIES'...>
```

generic — If a control's dynamic attribute is set to `generic`, the action is independent of objects selected on the page. There can be only one generic set of context and arguments. The action is associated with the context and arguments defined in the `actionmultiselect`

tag. For example, in the Webtop menubar.jsp page, the menu option **Copy** is enabled regardless of whether objects or folders are selected in the content frame.

genericnoselect — If a control's dynamic attribute is set to genericnoselect, the generic action will be disabled if any items on the page are selected. For example, in the Webtop menubar.jsp page, the menu option **Import** is disabled if an item in the classic view is selected.

Following is a table that summarizes the rendering of links based on the value of the dynamic attribute on an action control:

Table 3-3. State of a control cased on dynamic attribute value

Attribute Value	State in UI
generic	Control is always enabled
genericnoselect	Control is enabled when no item is selected in the content frame, otherwise it is disabled
singleselect	Control is enabled when there is a single object selected in the content frame, otherwise it is disabled
multiselect	Control is enabled when there is single object or multiple objects selected in the content frame, otherwise it is disabled. Requires dynamic.js in the JSP page.

Note: You must include /wdk/include/dynamicActions.js in the JSP page that contains an action control with the dynamic attribute value of "multiselect." For example:

```
<script language='JavaScript1.2' src='<%=Form.makeUrl(request,
  "/wdk/include/dynamicAction.js")%>'>
</script>
...
<dmfx:actionbutton dynamic='multiselect' ...action='checkout'>
</dmfx:actionbutton>
```

Combining dynamic control types — If you have an action control that is enabled for two dynamic conditions, you must create two controls. For example, you have a menu item or action button that is enabled when either one or no check boxes are selected. Create two controls. Set the dynamic attribute of one control, control A, to singleselect. Set the dynamic attribute of the other control, control B, to genericnoselect. Set the showifdisabled and showifinvalid attributes to false on both of the controls. Each control must have a different action ID. The action definition for control B can extend the action definition for control A. For example, the action for control B is defined similar to the following:

```
<action id="B" extends="A:custom/config/A_actions.xml"/>
```

Using dynamic action controls

The following example illustrates the use of the dynamic attribute to enable and disable menu items based on the state of checkboxes on the page. In the **New** menu (file_new_menu), the dynamic state of the menu item is based on the setting itself. The **New Document** (newdocument) menu item has a genericnoselect dynamic property. When this menu item is selected, the newdocument action will be executed if no objects in checkboxes are checked. The **New User** (newuser) menu item has a generic dynamic property. When this menu item is selected, the newuser action will be executed regardless of the state of checkboxes on the page.

In the **Edit** menu, the **Delete** menu item has a dynamic state of multiselect. The delete action will be executed on items on the page that are checked. The **Rename** menu item has a dynamic state of singleselect. This menu item will be enabled if one and only one checkbox is checked. The **View Clipboard** menu item will be enabled regardless of checkbox state.

```
<dmf:menugroup target='content' imagefolder='images/menubar'>
  <dmf:menu name='file_new_menu' nlsid='MSG_NEW'>
    <dmfx:actionmenuitem dynamic='genericnoselect' name='newdocument'
      nlsid='MSG_NEW_DOCUMENT' action='newdocument' showifinvalid='true' />
    <dmfx:actionmenuitem dynamic='generic' name='newuser'
      nlsid='MSG_NEW_USER' action='newuser' showifinvalid='true' />
  </dmf:menu>

  <dmf:menu name='edit_menu' nlsid='MSG_EDIT' width='50'>
    <dmfx:actionmenuitem dynamic='multiselect' name='delete'
      nlsid='MSG_DELETE' action='delete' showifinvalid='true' />
    <dmfx:actionmenuitem dynamic='singleselect' name='rename'
      nlsid='MSG_RENAME' action='rename' showifinvalid='true' />

    <dmfx:actionmenuitem dynamic='generic' name='viewclipboard'
      nlsid='MSG_VIEW_CLIP' action='viewclipboard' showifinvalid='true' />
  </dmf:menu>
  ...
</dmf:menugroup>
```

Controlling visibility

Two action control attributes govern a control's visibility (both static and dynamic control).

Show if invalid — Set showifinvalid to true if the control should be displayed when the associated action definition cannot be resolved by the configuration service (default = false).

Show if disabled — Set showifdisabled to true to display the control when one or more of the action preconditions return false. (default = true).

In addition to the `showifinvalid` and `showifdisabled` attributes, visibility is also determined by context. The following table shows the visibility of static and dynamic controls based on context:

Table 3-4. Control visibility based on context

	Static	Dynamic
Visibility is calculated when:	Control is initialized	Associated <code>actionmultiselect</code> control is rendered
Visibility is set when:	Control is rendered	Associated <code>actionmultiselectcheckbox</code> control is selected
Argument tags are nested within:	Control	Associated <code>actionmultiselectcheckbox</code> control. (Use this even for generic and <code>genericnoselect</code> dynamic actions.)

Argument tags are passed to the action from the action control or from the associated `actionmultiselectcheckbox` control. Argument tags nested within multiple selection controls such as `actionmenuitem` are ignored. For more information on passing arguments to action menu controls, refer to [Passing arguments to menus or dynamic action controls](#), page 180.

Controls that can be globally configured

Some controls support global configuration of all control instances through XML files. The global configuration can be overridden by configuration of a specific control on the JSP page. The following table describes controls that support global configuration. For more information on these controls, refer to *Web Development Kit Reference Guide*.

Table 3-5. Control global configuration

Control	Configuration
advsearch (various controls)	Configure value assistance (refer to Programmatic search value assistance , page 557), default match case (refer to Search query performance , page 350), size and dates (refer to Configuring search controls , page 145 in <code>/wdk/config/advsearchex.xml</code>)

Control	Configuration
breadcrumb	Configure breadcrumb controls to display or hide last leaf of breadcrumb and its style (refer to breadcrumb in <i>Web Development Kit Reference Guide</i>)
date controls	Configure dateinput, datevalueformatter, and datetime controls (refer to Configuring dates, page 177)
docbaseattributelist	Configure the display of attributes /webcomponent/config/library/attributes_*_docbaseattributelist.xml (refer to Attributelist configuration files, page 196)
docbaseobject	Configure (register) custom formatters and handlers for attributes when a docbaseobject is present on the JSP page /webcomponent/config/library/docbaseobjectconfiguration_*.xml (refer to Modifying the display and handling of attributes, page 395)
iconwell	Configure (add to) any component definition whose JSP page can display an iconwell (refer to iconwell in <i>Web Development Kit Reference Guide</i>)
paneset controls	Configure paneset controls to govern screen real estate. Refer to the paneset control in <i>Web Development Kit Reference Guide</i> for details.
richtexteditor, richtextdisplay	Configure allowable HTML tags for user input /wdk/config/richtext.xml (refer to Configuring rich text, page 190)
xforms	Customization not supported /wdk/config/xforms_config.xml

Hiding controls

You can configure a control on the JSP to hide it, rather than simply removing it from the JSP page. You may want to do this because you need to information that is set by the control but do not want to expose it to users. Or you may want to keep the control for possible use in the future.

To hide a control, use the CSS style on the table row that contains the WDK control. The following example hides the format selector and its label in the import JSP page:

```
<tr style="display:none;">
  <td>
    <dmf:label nlsid="MSG_FORMAT"/></td>
    <td class="defaultcolumnspacer">: </td>
    <td><dmf:datadropdownlist width="270" name="formatList" tooltipnlsid="MSG_FORMAT">
      <dmf:dataoptionlist>
        <dmf:option datafield="name" labeldatafield="description"/>
      </dmf:dataoptionlist>
    </dmf:datadropdownlist></td>
</tr>
```

Configuring dates

In addition to the configuration available on individual date controls, global date display can be configured by extending `/wdk/config/datetimecontrol_config.xml`. The following table describes date control configuration in this file:

Table 3-6. Global date control configuration elements

<code><dateinput></code>	Configures all instances of <code>dateinput</code> tag including those generated by the <code>doibaseattributelist</code> control. Can be overridden on an individual tag. Contains <code><default-year-from></code> , <code><default-year-to></code> , and <code><default_type></code> .
<code>.<default-year-from></code>	Sets the default beginning year in the date input dropdown list
<code>.<default-year-to></code>	Sets the default end year in the date input dropdown list

<datetime>	Sets the default date format type for all instances of the datetime control, including those generated by docbaseattributelist control. Overridden by the dateformat and timeformat attributes on the datetime control.
<datevalueformatter>	Sets the default date format type for all instances of the datevalueformatter control. Overridden by the type attribute on the datevalueformatter control.
.<default_type>	Specifies the type of Java DateFormat to apply (examples in parentheses). Valid values: short (12.13.52 or 3:30 pm) medium (Jan 12, 1952 3:30 pm) long (January 12, 1952 3:30:32 pm) full (Tuesday, April 12, 1952 AD 3:30:42 pm PST)

Configuring menus

In a desktop-style application design (classic or objectlist), the user selects operations and views from menus. Menus are displayed at the top of the object list. In a Web-style application design (streamline or drilldown), the user selects sets of operations or views through links or buttons or from the **More...** link. Both models are supported by WDK.

Actions can be grouped into menus, so that the set of actions available on a particular object type, user role, or other qualifier is specified in a single location. The <actionlist> element in an action definition lists the most commonly used actions that will be available for the specified scope. Within the action list definition, you can specify submenus using the <menu> element within a <moreactionsmenu> element. Menus contain one or more <menu> elements. Each menu is named by an id attribute of the <menu> element.

The menu element contains <action> elements that represent actions to be displayed in the menu. The id attribute of the <action> element must match the id attribute of an action definition in the application.

You can define more than one action list for the same scope within an application, provided that each action list has a different ID. This allows you to display different actions for the same object type in different pages.

When the JSP page includes the <actionlinklist> or <actionbuttonlist> tag, the action service uses the <actionlist> definition to build a JavaScript list of links or images that is appropriate for the object type. Menus and submenus are generated from

menu controls (<dmf:menu>). The menu items on the menu can be action items (<dmfx:actionmenuitem>) that require evaluation before the menu action is performed, or simple menu items (<dmf:menuitem>) that are performed for every user.

You can change the list of default actions for a given object type by editing the contents of the <actionlist> element in the action configuration file. For example, to move clipboard functionality from the displayed actions to the More actions link, move the element <action id="addtoclipboard" .../> from <actionlist> to one of the menus under <actionmenu>, such as <menu id="tools" ...>.

To create a menu for an object type:

1. In an action definition that is scoped to your object type, create the <menu> of available actions for the object type and specify submenus of actions using the <menu> element. Each action listed in the action menu must be defined somewhere in an action configuration file.
2. Add your action menu to the JSP page using the <dmf:menu> element. You must include the menu tag and specify the menu items. For example:

```
<actionmenu>
  <menu id="file" nlsid="MSG_FILE_MENU">
    <action id="delete" nlsid="MSG_DELETE" showifdisabled="false"/>
    <action id="export" nlsid="MSG_EXPORT"/>
  </menu>
</actionmenu>
```

3. Add a menu tag for each menu and an actionmenuitem tag for each menu item. For example:

```
<td>
<dmf:menu name="file_menu" nlsid="MSG_FILE" width="50">
  <dmfx:actionmenuitem dynamic="multiselect" name="file_delete"
    nlsid="MSG_DELETE" action="delete" showifinvalid="true">
  </dmfx:actionmenuitem>
  <dmfx:actionmenuitem dynamic="multiselect" name="file_export"
    nlsid="MSG_EXPORT" action="export" showifinvalid="true">
  </dmfx:actionmenuitem>
</dmf:menu>
</td>
```

In this example, the JSP will generate a **File** menu with the delete and export actions. The framework will look up the action definitions based on the object type of the objects selected in the UI. Menu items will be enabled or disabled based on the objects selected.

In addition to menus of actions, you can launch an action from an action button, action link, or action image control. The action service will look up the action definition based on the selected object type and then perform the same validation and launch process outlined above for menu actions.

Passing arguments to menus or dynamic action controls

Menu items do not pass arguments. The object selection tag `actionmultiselectcheckbox` passes the argument. This tag is used to pass all arguments, whether the action supports multiple selection, single selection, or no selection (`genericnoselect`). For example, the Webtop the home cabinet component displays objects in the user's home cabinet. When the user selects multiple objects in `doclist_body.jsp` (the UI for the home cabinet component) and then selects a menu item, arguments are passed from the `actionmultiselectcheckbox` control to the menu item's action.

Example 3-2. Passing an argument to a menu item

Add arguments that are required by the menu item action to `actionmultiselectcheckbox` tag. For example, in the Webtop `menubar.jsp` page, there is an `actionmenuitem` tag named `doc_vdm_freeze_assembly` that launches the `freezeassembly` action. The `freezeassembly` action definition requires an `isFrozenAssembly` param. This argument is passed to the action through the selectable object, namely, the `actionmultiselectcheckbox` tag in `doclist_body.jsp`:

```
<dmfx:actionmultiselectcheckbox name='check' value='false' cssclass='actions'>
  <dmf:argument name='objectId' datafield='r_object_id' />
  ...
  <dmf:argument name='isFrozenAssembly' datafield='r_has_frzn_assembly' />
</dmfx:actionmultiselectcheckbox>
```

To pass an argument to a custom menu item:

To pass an argument to your custom action, the argument must be specified as a parameter in your action definition and used by either your action precondition class, action execution class, or class of the component that is launched by the action. Then you must perform the following steps:

1. Add the action as a menu item.
2. Add the action arguments to the `actionmultiselectcheckbox` (this also works for `actionlinklist` tags). The following example adds `objectId` and `ownerName` arguments to `doclist_body.jsp` (additional arguments for other actions are also added to this tag):

```
<dmfx:actionmultiselectcheckbox ...>
  <dmf:argument name="objectId" datafield="r_object_id">
  </dmf:argument>
  <dmf:argument name="ownerName" datafield="owner_name">
  </dmf:argument>
  ...
</dmfx:actionmultiselectcheckbox>
```

3. These arguments are passed to a menu item in the Webtop `menubar.jsp`:

```
<dmfx:actionmenuitem dynamic="genericnoselect" name="file_newfolder"
  nlsid="MSG_NEW_FOLDER" action="newfolder" showifinvalid="true"/>
```

The arguments in `actionmultiselectcheckbox` are passed to the `newfolder` action. If you wanted to pass the same arguments to an action in a streamline view, add the arguments to the `actionlinklist` tag in the streamline UI, for example, `drilldown_body.jsp`.

Configuring tabs

Tabs are displayed across the top of the Web page. Each tab displays a different set of frames and JSP pages. Container components display tabs for each contained component. The start page for the container includes the `<dmf:tabbar>` tag. The entry tab is defined in the container component definition.

Configuring dropdown lists

Dropdown lists are used to provide a list of options that is either generated from a datafield or from fixed options. There are two types of dropdown lists; both can have a data source or fixed options:

- `dropdownlist`

Contains option tags or a `dataoptionlist` tag that provide options with data from the following sources:

- Values that are provided at design time in the JSP page, for example:

```
<dmf:option value="5" nlsid="Five"/>
```

- Values that are computed by the component class that uses the control, for example:

```
<dmf:option value='<%=Integer.toString(FormatList.DISPLAY_ALL_FORMATS) %>'
  nlsid='MSG_DISPLAY_ALL_FORMATS' />
```

- Values that are provided by a datafield, for example:

```
<dmf:datadropdownlist name="<%=FormatAttributes.FORMAT_RENDITION_LIST%>"
  tooltipnlsid="LABEL_RENDITION">
  <dmf:dataoptionlist>
    <dmf:option datafield="name" labeldatafield="name"/>
  </dmf:dataoptionlist>
</dmf:datadropdownlist>
```

You must set the mutable attribute to `true` (default = `false`) if you wish to alter the dropdown list programmatically from your component class.

- **datadropdownlist**

Contains option tags or a dataoptionlist tag that provide options with data in the following ways:

- Values that are provided by a datafield, for example:

```
<dmf:datadropdownlist name="<%=FormatAttributes.FORMAT_RENDITION_LIST%"
  tooltipnlsid="LABEL_RENDITION">
  <dmf:dataoptionlist>
    <dmf:option datafield="name" labeldatafield="name"/>
  </dmf:dataoptionlist>
</dmf:datadropdownlist>
```

- Values that are provided by options, one or more of which is provided by a datafield. For example:

```
<dmf:datadropdownlist name="<%=UserAttributes.USER_SOURCE%"
  tooltipnlsid="MSG_USER_SOURCE">
  <dmf:panel name="<%=UserAttributes.WINDOWS_DOCBASE_USER_SOURCE_CHOICES%">
    <dmf:option nlsid="MSG_USER_AUTHENTICATE_NONE"
      value="<%=Integer.toString(UserAttributes.USER_AUTHENTICATE_NONE)%">/>
  </dmf:panel>
  <dmf:dataoptionlist>
    <dmf:option datafield="pluginid" labeldatafield="pluginid"/>
  </dmf:dataoptionlist>
</dmf:datadropdownlist>
```

The datadropdownlist has a query attribute that can be used to provide options.

Example 3-3. Overriding a list of options

Some Documentum components provide a list of options, such as available object types in the import or newdocument components. You may wish to restrict this list of options. To do this, perform the following steps:

1. Extend the component definition and create a copy of the component JSP page in your custom directory.
2. Locate the dropdownlist tag in the JSP page. For example, in newdocument.jsp it is the following:

```
<dmf:datadropdownlist name="objectTypeList" onselect="onSelectType"...>
```

3. Replace the following lines:

```
<dmf:dataoptionlist>
<dmf:option datafield="type_name" labeldatafield="label_text"/>
</dmf:dataoptionlist>With
```

With these lines:

```
<dmf:option value="custom_type1" label="Custom Type 1"/>
<dmf:option value="custom_type1" label="Custom Type 2"/>
```

If you want to set the options dynamically in your component class based on some query or test, you can get the controls in the following way:

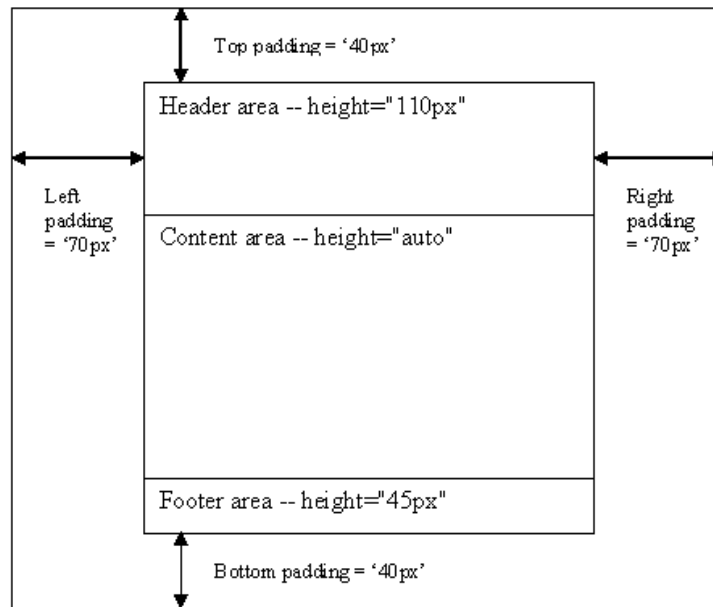
```
DropDownList dropdown = (DropDownList) get Control (
    DOCBASE, DropDownList.class);
if //test, use one set of options
{
    Option option1 = new Option();
    option1.set Value(value1);
    option1.set Label (value1);
    dropdown.addOption(option1);
}
else if //alternative, another set of options
{
    Option option2 = new Option();
    option2.set Value(value2);
    option2.set Label(value2);
    dropdown.addOption(option2);
}
```

Configuring scrollable controls

If your display is likely to be larger than the browser window, you can use scrollable panes within a paneset control. Header and footer panes, such as **OK** and **Cancel** buttons, will remain in view at all times.

The paneset control is the outer rectangle in the following illustration. The paneset will hide all other controls on the page. The paneset can contain nested panesets, with only one outermost paneset.

The paneset contains three panes: header, content, and footer. Each pane has an overflow attribute that governs scrolling. Each of the labelled parts is configurable as an attribute on the paneset or pane control:

Figure 3-1. Scrollable pane controls

For more information on configuring each control, refer to the reference information on the control.

Configuring databound controls

Databound controls read one or more values from a database or a repository and display the data in a formatted table (a data grid) or list. The data is retrieved from a JDBC connection, an existing in-memory recordset, or a DQL query. Other WDK controls can dynamically bind to the resulting data and display it.

All of the standard tag library controls can bind to data from a data container (refer to [Data support classes, page 390](#)). The `datafield` attribute of a control tag specifies the name or index of the column that contains the data. If the control receives its data from a `datafield`, the `datafield` attribute by default overrides the existing label or value attribute. In the following example, the `datafield` provides the label content, overriding the label control's label attribute (not specified here), and a `datafield` in a checkbox tag provides the Boolean value for full-text indexing:

```
<dmf:datagridRow>
  <td><dmf:label datafield='object_name' /></td>
  <td><dmf:checkbox datafield='a_full_text' />
</dmf:datagridRow>
```


Note: Any component that uses a databound control in a JSP page must establish a session and data provider for the control. Databound controls cannot be used on a JSP page that doesn't have a Documentum session. Refer to [Providing data to databound controls, page 186](#) for information on establishing a session and data provider.

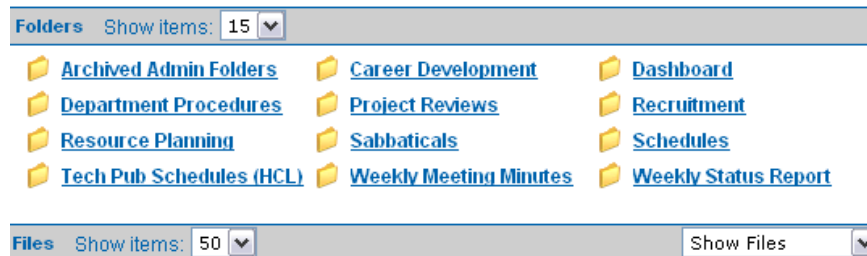
The following topics describe data binding:

- [Configuring data display, page 185](#)
- [Providing data to databound controls, page 186](#)
- [Configuring data sorting, page 188](#)
- [Configuring data paging, page 189](#)

Configuring data display

You can configure the number of columns in which your data is displayed using the `columns` attribute on a `datagridRow` tag. This is not the same display concept as the columns of data in the row. For example, in the drilldown (or Webtop streamline) UI, the list of folders, with an icon and a link to the folder name, is displayed in three columns:

Figure 3-2. Number of columns in a datagrid



To generate the columns shown above, the `datagridRow` tag uses the `column` element. The row contains two table cell tags that generate the icon and link:

```
<dmf:datagridRow name='objname' columns='3' width='33%'...>
  <td width=28>
    <dmfx:docbaseicon typedatafield='r_object_type'.../>
  </td>
  <td>
    <dmf:link ...onclick='onClickFolder' datafield='object_name'>
      <dmf:argument name='objectId' datafield='r_object_id' />
    </dmf:link>
  </td>
</dmf:datagridRow>
```

Providing data to databound controls

You need to provide a data source for databound controls. WDK controls that have a tag attribute "datafield" can display data that is provided by a databound container control such as datagrid and datadropdownlist or a provided by a data provider in the component class. The datafield attribute extracts the value for the named attribute from the data column that is named as the datafield.

The datafield overrides one of the control's properties. For example, a text control has a datafield attribute that overrides the label attribute if a data column is specified. By default, the "value" attribute of a control is overridden. In the following example, a data container control (datagrid) provides data to two databound controls: label and checkbox. The datafield attribute value on the databound controls match an attribute in the underlying recordset or query.

```
<dmf:datagrid name="mygrid" paged="true" pagesize="10" query="
  select object_name, a_full_text from dm_document">
  <dmf:datagridRow>
    <td><dmf:label datafield="object_name"/></td>
    <td><dmf:checkbox datafield="a_full_text"/></td>
  </dmf:datagridRow>
</dmf:datagrid>
```

A databound control tag must be placed within a databound container control such as datagrid, datadropdownlist, or datalistbox, or a control that extends one of these three data provider controls, in order to get access to the data. Set the datafield attribute on the JSP page to the name of the data column in the recordset or query. Some tag classes have more than one datafield. For example:

```
<dmfx:docbaseicon formatdatafield='a_content_type'
  typedatafield='r_object_type' linkcntdatafield='r_link_cnt'
  isvirtualdocdatafield='r_is_virtual_doc' size='16' />
```

The component class that contains databound controls in its JSP pages must initialize the controls with a data provider. In the following example from a component onInit() method, a datadropdown list is initialized:

```
DataDropDownList dropDownList = ((DataDropDownList) getControl(
  FORMAT_RENDITION_LIST, DataDropDownList.class));
dropDownList.getDataProvider().setDfSession(getDfSession());
```

To bind a control to a data source:

1. To access the data, place the control tag within either a databound container control (such as datagrid, datadropdownlist, or datalistbox) or a control that extends one of those container controls.
2. Set the datafield attribute in the JSP page to the name of the data column in the recordset or query. Some tag classes have more than one datafield. In the following example, the list of options is populated by the name datafield:

```
<dmf:datadropdownlist name="<%=FormatAttributes.FORMAT_RENDITION_LIST%>">
```

```

    <dmf:dataoptionlist>
      <dmf:option datafield="name" labeldatafield="name">
        </dmf:option>
      </dmf:dataoptionlist>
    </dmf:datadropdownlist>

```

The control class will use the data provider and data helper to extract the value for one of the databound control properties from the data column named as the value of the databound control datafield attribute.

Controls can retrieve any number of rows from a data provider. A single datagridrow tag will display all of the resulting values, each in a single row, unless the datapaging tag is applied.

When you write a control, you must decide which control property should be overridden by the datafield. (Some controls supply databinding to more than one property by creating additional databound tag properties such as "typedatafield".) Your control class implementation of setControlProperties() should call isDatafieldHandlerClass() to determine whether setControlProperties() was called from an extension class. If not, setControlProperties() should determine which property to override with the datafield value. Subclass controls can then either override the datafield property or use the super class implementation. The control should call resolveDatafield() on the super class, which binds to the appropriate column during rendering. For example (error-handling code omitted):

```

protected void setControlProperties(Control control)
{
    super.setControlProperties(control);
    StringInputControl input = (StringInputControl)control;

    // Set the control properties from the tag attributes
    // Ctrl value is overridden by the datafield and nlsid attributes
    if (getDatafield() != null && isDatafieldHandlerClass(
        StringInputControlTag.class) == true)
    {
        String strResult = resolveDatafield(getDatafield());
        input.setValue(strResult);
    }

    //datafield overrides NLS string
    else if (getNlsid() != null &&
        isNlsidHandlerClass(StringInputControlTag.class) == true)
    {
        input.setValue(getForm().getString(getNlsid()));
    }

    //If no datafield or NLS value, then set value from
    //some initialization value in your class
    else if (m_strValue != null)
    {
        input.setValue(m_strValue);
    }
}

```

Configuring data sorting

Data returned from a query is sorted automatically. All available data is read into memory and sorted based on locale sort rules provided by the Java framework. You can support sorting on a per-column basis with the `datasortimage` or `datasortlink` tag. The `datasortimage` tag renders an image that launches sorting. The `datasortlink` tag renders links that enable the user to sort the results by a column name. Each data sort tag must have a name that is unique among the sort tag names on the JSP page.

To sort data:

The `datasort` tags can be placed inside a `datagrid` tag. The tag is linked to the enclosing data container control by specifying a column name as one of its attributes.

1. Place the data provider control within a table cell. This tag will generate an HTML table. For example:

```
<td>
<dmf:datagrid name='permissiongrid' paged='true' pagesize='10'
  preservesort='false' cssclass='doclistbodyDatagrid' width='100%'
  cellspacing='0' cellpadding='0' bordersize='0'>
```

2. Define the sortable columns. In this example, the first row defines three sortable columns named 'object_name', 'owner_name', and 'description'. The column names are matched to `columnname` values in controls listed after the `datasortlink` tags:

```
<tr height='24'>
  <td>
    <dmf:datasortlink name='sort0' nlsid='MSG_OBJECT_NAME'
      column='object_name' cssclass='doclistbodyDatasortlink' />
  </td>
  <td>
    <dmf:datasortlink name='sort1' nlsid='MSG_OWNER_NAME'
      column='owner_name' cssclass='doclistbodyDatasortlink' />
  </td>
  <td>
    <dmf:datasortlink name='sort2' nlsid='MSG_DESCRIPTION'
      column='description' cssclass='doclistbodyDatasortlink' />
  </td>
</tr>
```

3. Place the data column tags within a `datagridRow` tag:

```
<dmf:datagridRow height='24' cssclass='contentBackground'>
  <td align="left">
    <dmf:label datafield="object_name"></dmf:label>
  </td>
  <dmf:columnpanel columnname="owner_name">
    <td>
      <dmf:label datafield="owner_name"></dmf:label>
    </td>
  </dmf:columnpanel>
  <dmf:columnpanel columnname="description">
    <td>
```

```

        <dmf:label datafield="description"></dmf:label>
      </td>
    </dmf:columnpanel>
  </dmf:datagridrow>

```

4. Close the data grid:

```
</dmf:datagrid>
```

Configuring data paging

The `paged` attribute on the `datagrid` control provides links that enable the user to jump between pages of data within the enclosing data container. You should page your data for better performance and display. If you set the `paged` attribute to `true`, the data provider or data container will render the appropriate links only if the provider has returned multiple pages of data from the query. The `datagrid` control is the only container that supports paging.

The `pagesize` attribute on the `datagrid` control sets the number of items to be displayed on a page. The default size is 10.

To support data paging:

1. Set the `paged` attribute on the data grid:

```
<dmf:datagrid name='<%=Messages.CONTROL_GRID%>' ...
  paged='true' pagesize='15'>
```

2. Add the paging controls in a header or footer row (at the beginning or end of your table, somewhere within the `datagrid` tag):

```
<!-- footer containing paging controls -->
<dmf:row height='5'>
  <td colspan=2 align=center>
<dmf:datapaging name='pager1' /> </td>
</dmf:row>
...

```

JSP fragment control

The JSP fragment control can include into a component page JSP fragments that are dispatched based on the client environment. To include a JSP fragment in a component page, add a `<dmfx:fragment>` control. The source of the fragment is specified as the value of the `src` attribute. An absolute path that begins with `"/`, such as `src="/wdk/fragments/modal/ModalContainerStart.jsp"` will always include the specified

fragment. A relative path, such as `src="modal/ModalContainerStart.jsp"` will dispatch a fragment based on runtime context.

Fragment bundles are defined in the application definition, for example, in `/custom/app.xml`. The `fragmentbundles` element contains the fragment bundles definition. The fragment bundle to use as a source for fragments for each client environment at runtime is specified as the value of the `<default-fragmentbundle>` element. In the following example, the client environment filter for AppConnectors specifies the default bundle as `appintg`. The optional `<base-fragmentbundle>` element specifies the base for lookup.

```
<filter clientenv="appintg">
  <default-fragmentbundle>appintg</default-fragmentbundle>
</filter>
<filter clientenv="not appintg">
  <default-fragmentbundle>webbrowser</default-fragmentbundle>
</filter>
<fragmentbundle>
  <name>webbrowser</name>
</fragmentbundle>
<fragmentbundle>
  <name>appintg</name>
  <base-fragmentbundle>webbrowser</base-fragmentbundle>
</fragmentbundle>
```

The fragment bundle lookup mechanism is similar to that for themes (refer to [How themes are located, page 126](#)). For example, based on the fragment bundle definition shown above, the lookup sequence for a JSP fragment tag whose `src` attribute has the value `"modal/ModalContainerStart.jsp"` will be the following for the Webtop application in the `appintg` (Application Connectors) `clientenv` context:

```
/webtop/custom/fragments/appintg/modal/
/webtop/webtop/fragments/appintg/modal/
/webtop/webcomponent/fragments/appintg/modal/
/webtop/wdk/fragments/appintg/modal/
/webtop/custom/fragments/modal/
/webtop/webtop/fragments/modal/
/webtop/webcomponent/fragments/modal/
/webtop/wdk/fragments/modal/
```

The fragment bundle processing is handed by the `FragmentBundleService` class. To trace problems with dispatching of fragment bundles, turn on the tracing flag `FRAGMENTBUNDLESERVICE..`

Configuring rich text

The data type `dmc_richtext` can be used as an attribute or to store an object. Rich text consists of HTML, including images and links. Rich text data is indexed in 5.3

Content Server. The control richtexteditor creates or modifies rich text, and the control richtextdisplay displays rich text.

The configuration file /wdk/config/richtext.xml configures the input and display of rich text attributes. (This is a different editor from the xforms rich text editor. It is installed as a DLL if enabled in app.xml.)

The following elements can be configured.

Table 3-7. Rich text configuration elements

Element	Description
<inputfilter>	Remove the comments from this element to use the RichTextInputFilter class, which processes images and links that are entered by the user
<outputfilter>	Remove the comments from this element to use the RichTextOutputFilter class, which processes image URLs for display in IE and Mozilla
<html_input>	Contains elements that govern HTML input
<allowed_tags>	Contains all HTML tags that are allowed within rich text input. Must include start and closing tag, for example, <td></td>.
<allowed_attributes>	Contains all attributes that are allowed within HTML tags. Attributes are represented as though they were HTML tags, for example, <href></href>.
<allowed_protocols>	Contains protocols that are permitted in links within rich text. For example, to prevent ftp URLs, remove <ftp></ftp> from the list.
<invalid_stylesheet_constructs>	Contains stylesheet constructs that are not permitted, such as those that contain external links. For example: <pre><div style="background: url('http://www.somewebsite.com/image/someimage.jpg');"> </div></pre>

Elements that configure the rich text editor are described in the table below. If no minimum version is provided, the browser will not be allowed to use the rich text editor.

Table 3-8. Rich text editor configuration elements (<editor>)

<ieminversion>	Minimum version of IE supported by editor
<mozillaminversion>	Minimum version of Mozilla supported by editor
<netscapeminversion>	Minimum version of Netscape supported by editor
<safariminversion>	Minimum version of Safari supported by editor
<iemaxversion>	Maximum version of IE supported by editor
<mozillamaxversion>	Maximum version of Mozilla supported by editor
<netscapemaxversion>	Maximum version of Netscape supported by editor
<safarimaxversion>	Maximum version of Safari supported by editor

Displaying and validating attributes

You can display individual attributes, both single-value and repeating, using one of the attribute controls. The `docbaseattributevalue` tag displays a single attribute value. The `docbaseattribute` tag displays an attribute value and attribute label. The `docbaseattributelist` control displays a list of attributes for a specific object type or other qualifier value.

You can apply special formatters and value handlers that apply to specific attributes or attribute types.

Attribute configuration is described in the following topics:

- [Single and repeating attributes, page 193](#)
- [Displaying lists of attributes, page 193](#)
- [Display of escaped HTML strings, page 201](#)
- [Configuring pseudoattributes, page 201](#)

Single and repeating attributes

You can display a single attribute value using a `<dmfx:docbaseattributevalue>` tag. You can display an attribute value with its data dictionary label using `<dmfx:docbaseattribute>`. Both `docbaseattributevalue` and `docbaseattribute` perform data dictionary validation of the value.

You can display multiple attributes for an object type using a `<dmfx:docbaseattributelist>` tag, which renders each attribute and its data dictionary label.

If a rendered single attribute is editable, a textbox is displayed for int, dbl, and string types, a checkbox is displayed for Boolean types, and a datetime control is rendered for date types.

If a rendered repeating attribute is editable, an **Edit** link that loads the repeating attribute editor is rendered.

The `docbasesingleattribute` and `docbaserepeatingattribute` component definitions can be extended and configured, so that you can present a different editing page for different object types.

Displaying lists of attributes

You can display lists of attributes using the `docbaseattributelist` control. The `docbaseattributelist` control eliminates the need to use a separate `docbaseattribute` control for each attribute to be listed in the UI. The list control allows you to display a different list of attributes for each component and for each user context such as role or object type. The attributes that are displayed are configured in Documentum Application Builder or directly in the `attributelist` configuration file, depending on which approach you choose.

You can use your own custom tags to display certain attributes or attribute types as generated by the `DocbaseAttributeListTag` class. You must register your custom tags in a `docbaseobject` configuration file. Refer to [Modifying the display and handling of attributes](#), page 395 for details.

The following topics describe how to use the control and configure a list of attributes for various contexts:

- [The attributelist control](#), page 194
- [Context-based attribute lists](#), page 195
- [Attributelist configuration files](#), page 196
- [Using data dictionary attribute lists](#), page 199
- [Supplying or overriding data dictionary attribute lists](#), page 200

The attributelist control

To display a list of attributes for a specific component or context, you must place a `<dmfx:docbaseattributelist>` control in a component JSP page. This control will generate a list of attributes that is defined in the data dictionary or in a WDK attributelist configuration file. A configuration setting in the attributelist configuration file tells the configuration service whether to read the list from the data dictionary or from the file itself.

The `docbaseattributelist` control has three required attributes and several optional attributes:

```
<dmfx:docbaseattributelist
  1name=list_name
  2object=object_name
  3attrconfigid=list_id
  4visiblecategory=category_name/>
```

1 Names the control. The control must be named to retain state during navigation.

2 Identifies the `docbaseobject` control for which the attributes will be displayed, for example:

```
<dmfx:docbaseobject name="obj"/>
<dmfx:docbaseattributelist name="attrlist"
  object="obj" .../>
```

3 Identifies the attribute configuration definition.

4 Comma-separated string that specifies the categories that are visible. Default = null (all categories are visible)

The value of the `attrconfigid` attribute in your list control must match the value of the `id` attribute on an attributelist configuration file. For example, in the `checkin` component JSP page `checkin.jsp`, you have a `docbaseattributelist` control as follows:

```
<dmfx:docbaseattributelist... attrconfigid="checkin">
</dmfx:docbaseattributelist>
```

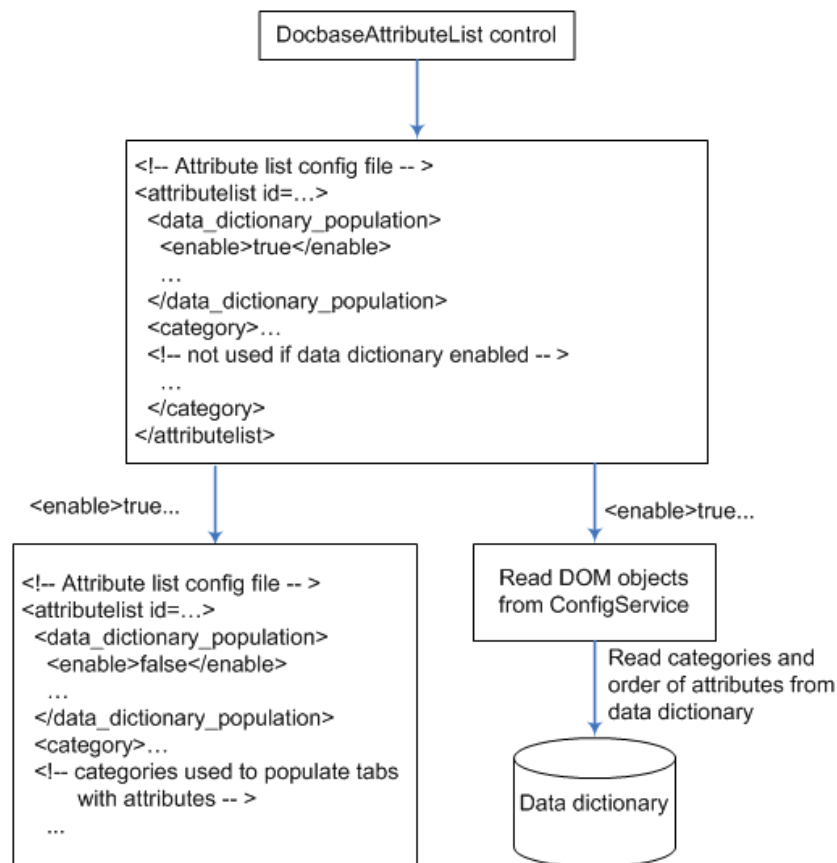
This matches the attributelist ID in the file `checkin_docbaseattributelist.xml`:

```
<attributelist id="checkin" ...>
```

The configuration file that is named in the `attrconfigid` attribute will determine the list of attributes that is displayed for the user's context and the source of the attribute list (data dictionary or XML file). Refer to [Attributelist configuration files, page 196](#) for a description of the attribute configuration file. Refer to [Context-based attribute lists, page 195](#) for a description of context-based list definitions.

You can configure the sets of attributes for display manually, bypassing the data dictionary, by setting `<data_dictionary_population>` to false in the attributelist configuration file. The two paths of attribute configuration are diagrammed below:

Figure 3-3. DocbaseAttributeList population



Context-based attribute lists

Attributes can be grouped into lists. Lists are further broken down into categories, which are displayed in the UI as tabs. Categories can also be displayed in the UI as separate sections on the same tab by setting the value of `<showpagesastabs>` to false in the attributes component configuration file.

Attribute display lists and categories are defined either in an XML configuration file (refer to [Attributelist configuration files](#), page 196) or in the data dictionary. You can define a different set of attributes to be displayed for each WDK-based component or user context.

If you use the configuration file to provide an attribute list, refer to [Attributelist configuration files, page 196](#) for details.

If you use Documentum Application Builder (DAB) to set up scopes and tabs (categories), refer to the DAB documentation. Each scope in DAB must match a qualifier in your application. For example, if you define a scope "type" in DAB, you must have the qualifier "type" enabled in your application.

The data dictionary scopes that you define in Documentum Application Builder might look like the following:

Table 3-9. Sample Documentum Application Builder scope definitions

Scope	Value
application (required)	Webtop
role	administrator, contributor
type	dm_document, custom_type

The configuration service matches the user's context to an attribute configuration file. For example, when an administrator is viewing the attributes for a dm_document object, the configuration service looks at the configurations in memory to find a definition that matches the context. It finds a definition for scope role="administrator", type="dm_document". The definition tells whether the attribute list should be read from the data dictionary or the configuration file. If data dictionary is specified, the configuration service must query the data dictionary of the current repository find a matching scope that was defined using DAB. The configuration service will then fetch the list of attributes that was defined in the data dictionary for that scope.

Refer to [Scope, page 52](#) for details on how to use the <scope> element in WDK configuration files.

Attributelist configuration files

The attribute configuration file is an XML file that performs several functions:

- Specifies whether the configuration service should read attribute lists from the data dictionary or from the configuration file. If the value of <data_dictionary_population>.<enable> is true, the data dictionary is used.
- Controls the display of attribute lists if they are not specified in the data dictionary (pre 5.2 repositories) or overrides the data dictionary display setting (<data_dictionary_population> = false)
- Tells the configuration service the name of the application that is calling for attributes, using the <ddscopes> element

Your attributelist configuration file specifies one or more scopes and scope values that match to scopes and scope values set up in the data dictionary, similar to the following. (You do not need to specify the application scope, because the configuration service adds the current application to the scope.)

```
<scope role="administrator", type="custom_type">
```

Each component that uses the docbaseattributelist control can specify its own configuration file with a different ID. This allows you to present a different set of attributes for each component that displays attributes. The attrconfigid attribute on the docbaseattributelist control is matched to the attributelist ID in the configuration file. For example, the attributes, import, and checkin components each have their own attributelist configuration file. Specifically, the checkin component UI contains a docbaseattributelist control whose attrconfigid value is "checkin." The configuration service finds the correct definition by looking for a configuration file with the following element:

```
<attributelist id="checkin" ...>
```

Tip: Extend an attributelist configuration file in the custom layer to control the display of attributes that are displayed for a qualifier or set of qualifiers in the application. When you extend an XML definition, you do not need to copy the entire contents of the base definition. If you define an element in your extended definition, you must copy all of the contents of that element if you wish them to be a part of your definition.

The configuration file has the following element hierarchy:

```
<config version='1.0'>
1<scope>
2<attributelist id=list_name>
  3<data_dictionary_population>
    4<enable>true | false</enable>
    5<ddscopes>
      <ddscope name="application">app_name</ddscope>
    </ddscopes>

    6<ignore_attributes>
      <attribute name=attribute_name/>
      ...
    </ignore_attributes>
    7<readonly_attributes>
      <attribute name=attribute_name/>
      ...
    </readonly_attributes>
  </data_dictionary_population>

  8<category id=category_name>
    <name><nlsid>NLS_key</nlsid></name>
    9<attributes>
      <attribute name=attribute_name/>
      <separator/>
      <attribute nameattribute_name/>
    </attributes>
  10<moreattributes>
```

```
        <attribute name=attribute_name/>
    </moreattributes>
</attributelist>
</scope>
</config>
```

- 1 If a scope is specified, the attribute lists defined in the <config> element apply only to user contexts that match the specified scope value.
- 2 Contains a defined list that is identified by the ID attribute.
- 3 Contains settings that specify the application name and turn on or off data dictionary population
- 4 Set to true to use the category information, order of attributes in a category, and order of categories from the data dictionary (Server version 5.2 and above). Set to false to use the values in the configuration file. The client display does not write back to the data dictionary. If enable is set to true, and the client connects to a 5.1 repository, the data dictionary settings will be merged with the configuration file settings .
- 5 The value of <ddscope> specifies a valid scope for the attributelist. Currently the only supported scope name is "application". The value of the <ddscope> application element must match a scope_value for the scope_class "application" that you have set up in the data dictionary. A value of "webtop" will be assumed unless another value is supplied for this element. Web Publisher uses the <ddscope> application value WebPublisher.
- 6 Contains <attribute> elements that specify attributes that should not be displayed in the UI.
- 7 Contains <attribute> elements to specify attributes that should be displayed as read-only in the UI but are editable in the data dictionary. (All standard server attributes that begin with a _ are modifiable, but you may wish to display them as read-only.) This readonly setting is applicable only for attributelists that are enabled to read from the data dictionary.
- 8 Defines a category. The id attribute is required for this element and must match the value of category_name for the attribute in the data dictionary for pre-5.2 repositories. The category definition is overridden by the data dictionary if the value of <enable> is true.
- 9 Contains <attribute> elements that specify attributes in the category. The value of <attribute> must correspond to an attribute defined for the type as specified in the <scope> element. You can generate a separator between attributes using the <separator/> element.

10 Contains <attribute> elements that specify a secondary list of attributes in the category. These attributes will be hidden in the UI and displayed only by a **Show More** link.

Using data dictionary attribute lists

To use the data dictionary lists of attributes, your docbaseattributelist configuration file must set the value of the <data_dictionary_population> element to true (the default). The runtime context will determine which set of attributes is displayed from the data dictionary. The configuration service looks up the appropriate docbaseattributelist configuration file whose scope matches the user context. For example, a user having the role of administrator logs on to Webtop and views the properties of a custom_type object. The configuration service looks for a definition with the attrconfigid named in the docbaseattributelist control. The service then matches the user context to a docbaseattributelist definition with the same ID and scope. In this example, the service looks for a configuration file whose scope is role="administrator", type="custom_type".

To the context, the configuration service adds the application name, which is specified in the docbaseattributelist definition as the value of <dds scope application=app_name>. The configuration service queries the data dictionary in the current repository and get the set of attributes defined for application='Webtop', role='Administrator', type='custom_type'. Any attributes that are named in the configuration file element <ignore_attributes> will not be displayed. Any data dictionary category that is named in the visiblecategory attribute of the docbaseattributelist control will be displayed. If the attribute is null, all categories defined for the scope are displayed.



Caution: Set up your scopes for displaying data dictionary values using Documentum Application Builder. If your object type has constraints, be sure to define a scope that displays the constrained attributes and then use that scope in your WDK-based attribute list scope for import and checkin. If you do not display constrained attributes for import and checkin, the validation process will fail.

The data dictionary for Content Server version 5.1 or lower has a category_name property that allows users to assign attributes to categories. Categories will be displayed as tabs or pages when they are displayed in a WDK-based application. Attributes with no category_name value will not be shown in the UI.

If you are connecting to a repository version 5.2 or above, the category_name attribute is not used. Categories (tabs) and scopes are set up in Documentum Application Builder (DAB). Refer to the DAB documentation for information on how to set up tabs and scopes.

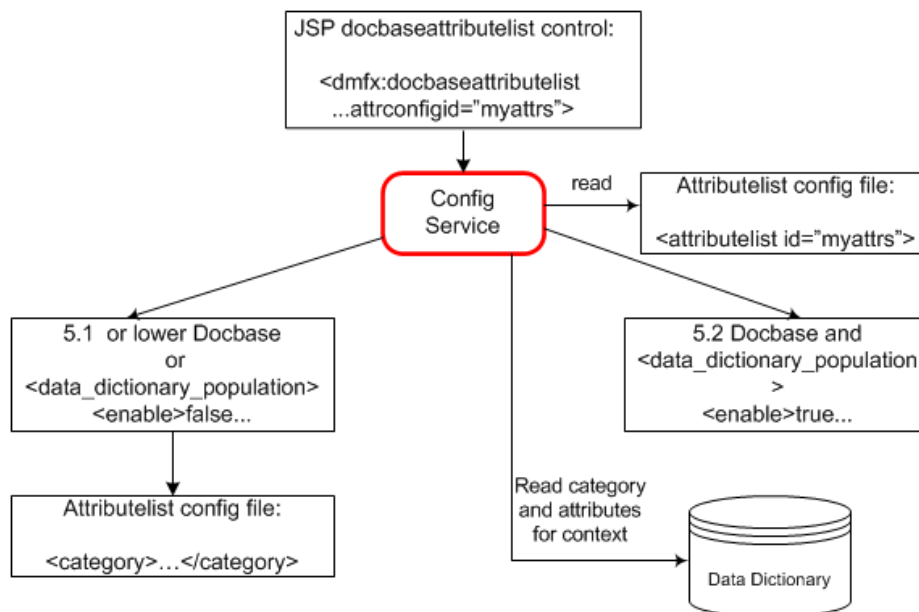
Supplying or overriding data dictionary attribute lists

If you set the value of `<data_dictionary_population>` to false, the categories of attributes will be read from the attributelist configuration file and not from the data dictionary. Use a `<category>` element to specify the attributes to be displayed in a tab.

If the data dictionary does not contain attributes associated with the Documentum object, all visible (`is_hidden=false`) attributes will be displayed. Use the `<ignore_attributes>` element to specify attributes that should not be displayed in the UI.

The process by which the configuration service determines the source for an attribute list is shown in the following diagram:

Figure 3-4. How the configuration service determines attribute list source



You can extend a definition using the same extends attribute that is used for other WDK configuration files. (Refer to [Extending XML definitions](#), page 51.)

Note: The attributes defined in the `<category>` elements in `attributes_docbaseattributelist.xml` will not be inherited by `import_docbaseattributelist.xml` or `checkin_docbaseattributelist.xml`. You must enumerate every attribute that you want to appear within each `<category>` element. This is a general principle for all configuration elements: If you extend a definition and change an element within a parent element, the parent element definition is overridden.

Categories (tabs or groups of attributes) that are defined in the `attributelist` configuration file can be displayed or hidden in any particular JSP by using the `visiblecategory` attribute of the `docbaseattributelist` control.

Display of escaped HTML strings

Prior to WDK 5.3, object attributes that contained HTML-reserved characters had to contain escape sequences in order to be displayed correctly in a browser. For example, a document attribute with the value "`<Work>`" would be displayed as "`<Work>`". (A document attribute with the value of "`<Work>`" could not be interpreted by the browser.)

To avoid the risk of cross-site scripting in which a user could enter malicious JavaScript as an attribute value, attribute values are now displayed exactly as they are stored, so that document attributes with a value of "`<Work>`" will be displayed as "`<Work>`", and document attributes with a value of "`<Work>`" will be saved and displayed as "`<Work>`".

You can change the default encoding to false by setting the value of the `<labelproperties>.<encodelabel>` element in your custom `app.xml` file. In that case, document attributes with a value of "`<Work>`" will be saved and displayed as "`<Work>`".

You can override the global setting in `app.xml` by setting the value of the `encodeattribute` to true or false in individual label controls. For example: `<dmf:label datafield="some_datafield" encodelabel="false"/>`. Note that if you turn off the encoding either globally or locally you put your application at risk for cross-site scripting.

The encoding is accomplished behind the scenes by encoding (escaping) the characters in the attribute so they are interpreted correctly by the browser. For example, "`<Work>`" is converted to "`<Work>`" which the browser interprets as "`<Work>`".

For information on methods to encode attributes and other user input, see [Formatting and escaping rendered HTML, page 366](#). For information on configuration of `app.xml` to detect URL parameters that are susceptible to cross-site scripting, refer to `<requestvalidation>` element, [page 71](#).

Configuring pseudoattributes

Pseudoattributes are defined within the WDK application and are displayed and handled within the application. For example, WDK defines the pseudoattribute type `RichText`. Any attribute of this type can be defined within the application, and those attributes will be displayed and edited using the rich text editor. The attribute value is handled by the

rich text attribute classes, which invoke the rich text BOF service to get and set rich text content associated with the sysobject.

A component that displays or edits the rich text attribute must call the rich text service in the component class. For information on creating a pseudoattribute, refer to [Creating custom pseudoattributes, page 431](#).

Validating user input

A validation control checks one input control for a specific type of error condition and displays a message if an error is found. Control input can be validated by more than one validation control.

Several validator controls are provided in WDK to validate data input. For more information on configuring individual validator controls, refer to *Web Development Kit Reference Guide*. For examples of using validators, refer to [Configuring validators, page 308](#).

The following topics describe validation and value assistance:

- [Validator controls, page 202](#)
- [Validating an object and its attributes, page 204](#)
- [Using value assistance, page 204](#)
- [Implementing non-data dictionary value assistance, page 205](#)

Validator controls

Validation on controls is performed when a form is submitted. Validation errors do not prevent a control event from being fired. The component that uses validation controls must implement error handling for validation errors.

When an input control needs validation, add the validator to the JSP page and assign values to two attributes:

- `controltovalidate`
Identifies the target control.
- `errormessage`
Specifies the message that will be shown for validation failure. This attribute can be replaced by the `nlsid` attribute, to specify the ID of a localized error message.

Some validation controls have additional attributes that configure the validation parameters.

By default, validation is performed on all validated controls in the form when a server-side action event is fired on a form. You can override form validation by adding the webform tag attribute "validation" and setting it to false. (By default this attribute has a value of true.) You may want to do this if validation is slowing down the UI redraw or if all events do not need validation. For example:

```
<dmf:webform formclass="com.documentum..."
  validation="false"/>
```

If an input control contains a null or empty value, it is assumed to be valid. You can use the `requiredFieldValidator` control to check for a null or empty value. You can combine `RequiredFieldValidator` with other validators to ensure that a valid value is provided and provide a different error message for each validation failure.

Input mask validator

The input value that is tested by an input mask validator control must match the specified input mask pattern. The mask accepts a string of characters with the following notations:

- #: Numeric characters
- &: All characters
- A: Alphanumeric characters only
- ?: Alphabet characters only
- U: Upper case alphabetic characters only
- L: Lower case alphabetic characters only

To escape a masked character for use as a literal member of the mask string, use a double slash ("`\\`") before the character. Here are some examples of popular masks:

Name	Mask	Example
Date	##/##/##	12/24/95
Time	##:## UU	12:35 AM
SSN	###-##-####	148-92-1532
Phone	(###) ###-#### [####]	(919) 933-0863 [7]
Zip code + 4	#####-####	27858-1203
First Name	??????????	Aloysius

Validating an object and its attributes

Repository object and attribute controls display data dictionary information for a selected object, such as labels and values. Use the tags `docbaseobject` and `docbaseattribute`, which embed validation controls so that Documentum objects are automatically validated when validation is turned on.

The `docbaseattribute` tag class calls `docbaseattributelabel` to render a label for the value and `docbaseattributevalue` to render the read-only or editable display of the attribute. The type of control that is rendered for display is based on the attribute datatype (for example, checkbox, date/time, textbox).

To use and validate a repository object:

Use the `docbaseobject` tag in your JSP page to display attributes of a Documentum object. The following example sets the object and then validates and displays attributes for the object:

1. Add a `docbaseobject` tag. For example:

```
<dmfx:docbaseobject name="myobject"/>
```

2. Add a `docbaseattribute` tag to be validated and displayed. For example:

```
<dmfx:docbaseattribute  
  object="myobject" attribute="r_object_type" readonly="true"/>
```

Using value assistance

The `docbaseattributevalue` and `docbaseattribute` controls detect the possible values in the data dictionary for a Documentum object attribute. If there is no value assistance in the data dictionary for an attribute, a simple text box is generated for editing the value of the attribute. If value assistance is tied to the attribute, the control generates a list of suggested values. The type of list that is generated is dynamically determined, based on the type of attribute:

- Non-repeating attribute, closed-end
A list values is presented for selection in a drop-down list control.
- Repeating attribute, closed-end
A link is presented. The link opens a JSP page with dictionary-backed selections for the attribute
- Non-repeating attribute, open end
A list box is presented for selection or for adding a value.
- Repeating attribute, open-end

A link is presented. The link opens a JSP page with dictionary-backed selections for the attribute as well as an editable text field.

For example, you have two `docbaseattributevalue` controls for the "day" and "chore" attributes. You have set up the list of valid values of chore in the data dictionary to depend on the value of day. When the drop-down list value for day changes, the drop-down list for chore is repopulated. Changes to other controls on the page that do not represent related attributes will not cause a page refresh.

To enable validation on a repository attribute, you must set the validation attribute of the webform tag to true in the parent form. If you set validation to false, validation is turned off for all controls inside the container:

```
<dmf:webform validation="true"/>
```

For information on setting up value assistance outside of the data dictionary, refer to [Implementing non-data dictionary value assistance, page 205](#).

Implementing non-data dictionary value assistance

Repository attributes are displayed with value assistance if value assistance has been set up in Documentum Application Builder. The `docbaseattributevalueassistance` control provides value assistance that is not based on data dictionary. Set the `docbaseattributevalueassistance` attribute of `docbaseattribute` or `docbaseattributevalue` control to use your custom value assistance list, and then provide the value assistance values using `docbaseattributevalueassistance` controls in the JSP or by setting the values in the component class.

You can use non-data dictionary based value assistance if you have not set up value assistance in the DocApp. If a particular attribute contains data dictionary based value assistance, the values specified in the `docbaseattributevalueassistance` control will be ignored.

For more information on each non-data dictionary value assistance control, refer to *Web Development Kit Reference Guide*.

To set up value assistance in the JSP page:

1. Set the `docbaseattributevalueassistance` attribute on the `docbaseattribute` or `docbaseattributevalue` tag to the name of a `docbaseattributevalueassistance` control to specify non-data dictionary assistance. For example:

```
<dmfx:docbaseattribute object="obj" attribute="keywords"
  docbaseattributevalueassistance="my_resultset" size="48"
  cssclass="defaultLabelStyle"/>
```

2. Add values using the `docbaseattributevalueassistance` and `docbaseattributevalueassistancevalue` tags. Set the attribute `islistcomplete` to true

for a closed list of values and false for an open-ended list. The following list is open-ended, so the user can add values that are not in the list:

```
<dmfx:docbaseattributevalueassistance name="my_resultset" islistcomplete="false">
  <dmfx:docbaseattributevalueassistancevalue label="Value A" value="valA"/>
  <dmfx:docbaseattributevalueassistancevalue label="Value B" value="valB"/>
</dmfx:docbaseattributevalueassistance>
```

To set up value assistance in the component class:

1. Set the `docbaseattributevalueassistance` attribute on the `docbaseattribute` or `docbaseattributevalue` tag to the name of a `docbaseattributevalueassistance` control to specify non-data dictionary assistance. For example:

```
<dmfx:docbaseattribute object="obj" attribute="keywords"
  docbaseattributevalueassistance="my_resultset" size="48"
  cssclass="defaultLabelStyle"/>
```

2. Add an empty `docbaseattributevalueassistance` tag whose values will be supplied by the component class. For example:

```
<dmfx:docbaseattributevalueassistance name="my_resultset" islistcomplete="false">
</dmfx:docbaseattributevalueassistance>
```

3. Add values in your component class:

```
DocbaseAttributeValueAssistance attributeVA =
    (DocbaseAttributeValueAssistance) getControl(
        "my_resultset", DocbaseAttributeValueAssistance.class);
attributeVA.setMutable(true);
```

```
DocbaseAttributeValueAssistanceValue attributeVAValue =
    new DocbaseAttributeValueAssistanceValue();
attributeVAValue.setLabel("Value A");
attributeVAValue.setValue("valA");
attributeVA.addValue(attributeVAValue);
```

```
DocbaseAttributeValueAssistanceValue attributeVAValue2 =
    new DocbaseAttributeValueAssistanceValue();
attributeVAValue2.setLabel("Value B");
attributeVAValue2.setValue("valB");
attributeVA.addValue(attributeVAValue2);
```

Working with images and icons

You can specify the path to individual images or icons used in controls with the `imagefolder` attribute. If the `imagefolder` attribute starts with `http:` or `https:`, it is handled as a complete URL. If the `imagefolder` attribute starts with a forward slash (`/`), it is handled as a path relative to the virtual root (for example, to the root of the WAR). If

the `imagefolder` attribute has no prefix, the `imagefolder` is handled as a path relative to the current theme directory.

Button, tab bar, and label controls can render themselves with or without images.

All graphics in the `/images` and `/icons` directories must have an entry in an accessibility resource file to support accessibility. The NLS string is displayed as an HTML `alt` attribute value in browser mouseover. For more information, refer to [Image accessibility strings, page 587](#).

Icon controls

Icon controls resolve the state of the icon and the image file that is displayed, based on repository attributes. If the type or format is not databound or an image is not found, the icon resolves to `t_unknown_16.gif` or `t_unknown_32.gif`. You can also set the state programmatically.

The following icon controls are provided in the tab libraries:

- `docbaselockicon`
Displays a lock if the object is checked out.
- `docbaseicon`
Displays an image based on document format or document type.
- `docbasepriorityicon`
Displays an icon representing the task priority in a user's inbox.

Using icons

To use an icon for a custom type:

1. Prepare two icons for the custom type in GIF format: one sized at 16x16 pixels, the other at 32x32 pixels.
2. Name the icons with the prefix "t_" and suffix "_16" or "_32". The name between these two strings must be the type name. For example, for your custom type `acme_sop`, your image files would be named `t_acme_sop_32.gif` and `t_acme_sop_16.gif`.
3. Place the two image files in `custom/theme/documentum/icons/type`. For information on the theme directory structure, refer to [Creating a theme directory, page 124](#).

To replace an image or icon on a single JSP page:

1. Open the JSP page and locate the button or icon.

2. Enter a new value for the imagefolder attribute on the button or icon. The path must be relative to the theme folder in which the graphic is located, with no leading slash.

To replace an image or icon across your application:

1. Create a custom style sheet, as described in [Modifying a style sheet, page 130](#).
2. Create a new class with a path to your image. For example:

```
.removeButton { BACKGROUND-COLOR: transparent;  
BACKGROUND-IMAGE: url('../images/removebg.gif') }
```
3. Add your image to the theme directories for which the image will be used. (You must provide your image for all themes, in the /custom/theme/*theme_name*/images directory.)
4. Reference your cssclass whenever a remove button is used. For example:

```
<dmf:button nlsid = "MSG_REMOVE" cssclass='removeButton' />
```

Working with tooltips

A tooltip is displayed by mouse hover in the browser. The Control class supports a tooltip, tooltipnlsid, and tooltipdatafield, but not all controls expose a tooltip in the tag library. This means that you can set the tooltip string, or NLS key, or datafield programmatically for any control. To use the tooltip attributes, the attribute must be present in the tag library descriptor.

Some of the controls that expose tooltip as a configurable attribute on the JSP tag include the following: link, text, label, button, checkbox, option, tab, and many others. Others such as button, datagridRow, and actionbutton expose support for the tooltipdatafield. Two text formatters have a showastooltip attribute that allow you to configure whether the entire contents should be displayed as a tooltip or truncated.

A tooltip, if configured on the JSP page or set by the container class, is rendered as a title attribute on an HTML element. A tooltip value is overridden by a tooltipnlsid, if present. Both are overridden by a tooltipdatafield, if present.

Controls that are rendered by DocbaseAttributeList controls do not provide access to the tooltip, either by configuration or programmatically.

Note: When the user has turned on accessibility mode, tooltips for images are not displayed. Instead, alt text is rendered for the reader. For more information on this accessibility feature, refer to [Configuring accessibility, page 310](#).

Configuring Actions

Any action-enabled control can launch an action. The action named as the value of the action attribute for the control must match an action ID in an action definition file. When a control launches an action, or an action launched by URL, the WDK framework searches for the action definition by action ID. Refer to *Web Development Kit Reference Guide* for the action attributes that configure action controls.

Actions can also be called by URL or from a component class.

The following topics describe action configuration and implementation:

- [What actions do, page 211](#)
- [How to launch an action, page 212](#)
- [Generic actions using LaunchComponent, page 215](#)
- [Action configuration file, page 215](#)
- [Precondition permissions, page 218](#)
- [Hiding an action for subtypes, page 218](#)

The common action definition settings and action-specific settings are described in *Web Development Kit Reference Guide*.

What actions do

An action is an operation that is typically invoked when a user interacts with the UI. You can also use an action to launch an operation when the action doesn't require a UI, such as the logout action.

The action service determines whether a user can perform an action based on preconditions. This ensures that buttons, links, menu items and tabs are active only if the related action can be performed in the current context of the UI. If the action has no precondition, such as the login action, it is executed for every context. Action preconditions are specified in the action definition and implemented in a precondition class. Actions are executed by an execution class, which is also specified in the action definition.

Actions are defined in an action configuration file (refer to [Action configuration file, page 215](#)). Actions can inherit and customize an action definition from another application layer in the same way that components and applications inherit their definitions. You can also override elements in the parent definition. For more information on inheritance, refer to [Extending XML definitions, page 51](#).

The action can be launched from the UI, from a repository operation, or from the action dispatcher by URL:

- UI: Actions can be launched when a user initiates a UI event such as opening a dialog window, navigating, or clicking a link, button, list item, or menu item
- repository operation: Actions can be launched as part of a component operation such as checkin, checkout, or import
- Action dispatcher: Actions can be launched by a URL. Refer to [How to launch an action, page 212](#) for details.

Actions can be role-based. Refer to [Chapter 16, Customizing Roles](#) for further information on role-based actions.

How to launch an action

Actions can be invoked in the following ways:

- URL to the action dispatcher servlet

The action dispatcher servlet operates similarly to the component dispatcher. Use a URL in the following format, substituting the actual application name and action name:

```
/my_application/action/action_name[?action_arguments]
```

For example:

```
http://myserver:8080/webtop/action/view?objectId=0901d8ab80015b6b
```

Note: URLs in JSP pages must have paths relative to the Web application root context or relative to the current directory. For example, the included file `<%@ include file='doclist_thumbnail_body.jsp' %>` is in the same directory as the including file. The included file `<%@ include file='/webcomponent/navigation/drilldown/drilldown_body.jsp' %>` is in the /webcomponent subdirectory of the Web application.

- Action-enabled controls

Specify the action name as the value for the "action" attribute of a control such as `actionlist`, `actionimage`, `actionlink`, or `actionmenuitem`.

- Component method

To launch an action from a component method, call the action service and pass the action name in a component class. For example:

```
public void onClickObject(Control control, ArgumentList args)
{
    try
    {
        ActionService.execute("view", args, getContext(), this, null);
    }
}
```

- Startup action in the application URL

You can log into an application with a URL that specifies the startup action to perform after login. The action's required parameters and their values must be in the URL. You can also pass optional parameter values in the URL. In the following example, the application opens with a login dialog and then presents the results of the query in the URL (line break inserted for display purposes):

```
http://myserver/webtop/component/main?startupAction=search&queryType=string&query=
some_query_string_here
```

Adding action controls to a JSP page

Many of the WDK controls such as buttons, menu items, or links are action-enabled. Action-enabled controls are automatically hidden or disabled if the associated action is not resolved or the precondition is not met. For more information on the individual controls, refer to [Action-enabled controls, page 171](#).

These action controls, when selected, invoke an action that is defined in the action attribute of the control tag. The parameters and context for the action are specified using the `<argument ...>` tag. For example:

```
<dmfx:actionlink cssclass='actions' name='renamefolderbtn'
    nlsid='MSG_RENAME' action='rename'>
    <dmf:argument name='objectId' datafield='r_object_id' />
</dmfx:actionlink>
```

Note: You must include `/wdk/include/dynamicActions.js` in the JSP page that contains a dynamic action control. For example:

```
<script language='JavaScript1.2' src='<%=Form.makeUrl(request,
    "/wdk/include/dynamicAction.js")%>'>
</script>
```

Passing arguments to actions

You can pass required or optional arguments from a control on a JSP page to an action. When an action is launched from an action control, add your arguments to the action

control using a `<dmf:argument>` or `<dmfx:argument>` tag. The argument can pass a value using the value attribute, the datafield attribute (which overrides the value attribute), or the contextvalue attribute, which overrides the datafield attribute.

Argument values can be hard-coded specifically to the preconditions or execution class by adding an `<arguments>` element. The following example from the Web Publisher action definition `workflowstatusclassic` adds an argument and value that is used by the execution class:

```
<execution class="com.documentum.wp.action.LaunchWpComponent">
  <arguments>
    <argument name='wpcontext' value='wpdefaultmyworkflow' />
  </arguments>
  <component>workflowstatusclassic</component>
</execution>
```

You can pass arguments from an action to a component. All defined parameters in the action definition are passed to the component from the action. The `objectID` argument value is always passed by the `LaunchComponent` execution class, so you do not need to explicitly pass this argument when you are using `LaunchComponent`.

Note: Arguments cannot be nested within dynamic action controls having the attribute value `dynamic=singleselect` or `dynamic=multiselect`, because the arguments are passed by the selection control (checkbox).

Example 4-1. Passing multiple selection values to an action

In the following example from `aclist.jsp`, two arguments are passed to all actions that support multiple selection on the JSP page:

```
<dmfx:actionmultiselect name="multiacl">
  <dmf:argument name="type" value="dm_acl"></dmf:argument>
  <dmf:argument name="objectId" value="newobject"></dmf:argument>
  ...
</dmfx:actionmultiselect>
```

Example 4-2. Passing a datafield to an action

In the next example from the same JSP page, `datafield` and `type` arguments are passed to an `actionimage` control. The `datafield` argument must be passed in a `<dmfx:argument...>` control:

```
<dmfx:actionimage nlsid="MSG_PROPERTIES" name="propImage"
  action="properties" src="icons/info.gif">
  <dmfx:argument name="objectId" datafield="r_object_id">
  </dmfx:argument>
  <dmf:argument name="type" value="dm_acl"></dmf:argument>
</dmfx:actionimage>
```

Example 4-3. Passing context values to an action

The datafield attribute on an action tag must correspond to a datafield for the selected object type. The context value can be supplied from the JSP page or from the component class. In the following example, a context value is set by the component class as follows:

```
boolean canAcquireTask = (state != ITask.DF_WF_TASK_STATE_PAUSED
    && state != ITask.DF_WF_TASK_STATE_ACQUIRED && state !=
    ITask.DF_WF_TASK_STATE_FINISHED);
context.set("canAcquireTask", String.valueOf(canAcquireTask));
```

The context value is then passed as an action button argument in the JSP page:

```
<dmfx:actionbutton ...contextvalue='canAcquireTask' />
```

The context value is then acquired in the action precondition class as follows:

```
if((strCanAquire = args.get(
    "canAcquireTask")) != null && strCanAquire.length() > 0
    ...
```

Generic actions using LaunchComponent

Generic actions are not based on object type or any other scoped qualifier. They use the LaunchComponent execution class to launch a specified component. If the generic action definition does not contain precondition elements, then the action is always executed. Generic actions are defined in `/wdk/config/actions/generic_actions.xml`. For more information on the LaunchComponent class, refer to [LaunchComponent execution classes, page 496](#).

Action configuration file

Actions are defined in XML configuration files. The configuration file contains one or more action definitions within `<action></action>` elements. The action definition provides the action with its context sensitivity. The action configuration settings are read into memory and retrieved by the configuration service. Definitions are cached for performance optimization.

Action definitions can be extended in the same way that application and component definitions are extended. Refer to [Application layer inheritance, page 42](#) for more information.

All action definitions have the following form. *Italics* denotes user-defined content. An asterisk (*) shows elements that can be repeated.

```
<config>
1<scope qualifier_name*=qualifier_value>
2<action id=action_id>
```

```

3<params>
  <param name=param_name required=true|false>
  </param>*
</params>

4<preconditions
  5<precondition class=action_precondition_class>
    6<role>role_name</role>
    7<argument>argument_name</argument>
    8<custom_precondition_element/>*
  </precondition>*
</preconditions>]

9<execution class=action_execution_class>
  10<permit>permit_level</permit>
  11<olecomponent enabled="true_or_false"/>
  12<navigation>jump_type</navigation>
  13<component>component_name</component>
  14<container>launch_container</container>
  15<custom_execution_element>*
    16<arguments>
      <argument name='argName' value='argValue' />*
    </arguments>
</execution>
</action>
</scope>
</config>

```

- 1** Defines the context for the action. Contains one or more primary elements and zero or more attributes. The scope is implemented by a qualifier such as object type, user role, or repository name. The attributes of the scope apply to the primary elements within the scope. An empty scope element applies to all conditions. For example, <scope type=dm_folder> applies to folders only, but <scope> applies to all objects.
- 2** An action is resolved by the ID attribute on the action element.
- 3** Defines parameters (<param>) that are passed to the precondition and execution class. <params> contains one or more elements. Attributes of <param>: name (string) and required (true | false).
- 4** (Optional) Contains one or more <precondition> elements that define preconditions for the action.
- 5** Contains zero or more custom-defined elements whose values are passed to the precondition class. The class attribute value specifies a fully qualified class name of a class that implements IActionPrecondition. The same class is sometimes used for preconditions and execution.
- 6** Determines whether the user has the required role. Valid values are role names defined in the repository.

7 This element is used with the precondition class `ArgumentNotEmptyPrecondition`. The value names a parameter contained within the `<parameters>` element that cannot be empty or null.

8 Custom precondition elements can be children of the `<precondition>` element. Precondition elements are defined by the precondition class.

9 (Required) Defines an execution class that implements `IActionExecution`. Attribute "class" has a fully qualified class name as its value. Contains zero or more user-defined child elements. Sometimes the same class is used for preconditions and execution.

The `LaunchComponent` class can be used to launch a component that will perform the action after preconditions are met. When `LaunchComponent` is used, you can add the optional child elements `<navigation>`, `<component>`, and `<container>`.

10 Determines whether the user has the required permission on the object. This element is used with the `LaunchComponentWithPermitCheck` execution class. Valid values are:

```
DELETE_PERMIT | WRITE_PERMIT | VERSION_PERMIT |
RELATE_PERMIT | READ_PERMIT | BROWSE_PERMIT | NONE_PERMIT
```

11 Specifies whether the launched component class is able to process compound documents. This element is supported by the `LaunchComponentWithPermitCheck` execution class and is valid only for that class or a subclass of that class. WDK components do not process compound documents.

12 Specifies the type of navigation from the current component to the component being launched. Valid values: `jump` | `returnjump` | `nested` (default).

13 Specifies the component to be launched. If none is specified, the default value is a component with the same ID as the action. Can contain `<arguments>` element which itself contains `<argument>` names and values that are passed to the component.

14 Specifies the container that will launch the component.

15 Specifies custom elements whose values are read by the execution class.

16 Specifies arguments to be passed to a component or container when you use the `LaunchComponent` execution class or a class that extends `LaunchComponent`. The argument tag has the following syntax:

```
<execution class='com.documentum.web.formext.action.LaunchComponent'>
  <component>component-id</component>
  <container>container-component-id</container>
  <arguments>
    <argument name='arg-name' value='arg-value'>
  </arguments>
</execution>
```

Precondition permissions

To use a permission precondition for an action, use the execution class `LaunchComponentWithPermitCheck` in the package `com.documentum.web.formext`. In the action definition, add a `<permit>` element as a child of the `<execution>` element. Valid values for `<permit>` are:

```
DELETE_PERMIT | WRITE_PERMIT | VERSION_PERMIT | RELATE_PERMIT |  
READ_PERMIT | BROWSE_PERMIT | NONE_PERMIT
```

Hiding an action for subtypes

If an action definition is scoped to a specific object type, the definition applies to subtypes of the type. You may wish to turn off the action for certain subtypes. To do this, create a scoped definition for the subtype and use the `notdefined` attribute. In the following example, the checkout action does not apply to folders:

```
<scope type='dm_folder'>  
  <action id="checkout" notdefined="true"></action>  
</scope>
```

The type "foreign" is a pseudo-type defined with WDK that is assigned by the `DocbaseTypeQualifier` class to reference objects. A list of actions scoped to the foreign type has the `notdefined` attribute set to true, so that those actions cannot be performed on reference objects. This list of undefined actions is combined with a dynamic filter on the action definition so that foreign objects display an invalid action message when the user selects the action. Refer to [Dynamic component launching, page 498](#) for details on the filter.

Configuring Components

A component consists of a component definition within an XML configuration file, one or more layout JSP pages, and a component behavior class. The component definition configures the behavior of a component.

For information on configuring specific WDK components, refer to *Web Development Kit Reference Guide*.

Components are described in the following topics:

- [Component features, page 219](#)
- [Component configuration file, page 221](#)
- [Component inheritance \(extends\), page 224](#)
- [Component scope, page 225](#)
- [Hiding components, page 227](#)
- [Hiding component features, page 228](#)
- [Configuring data columns, page 229](#)
- [Component layout \(JSP pages\), page 233](#)
- [Component navigation, page 237](#)
- [Component operations on foreign objects, page 242](#)
- [Presubmission client events, page 242](#)
- [Configuring containers, page 243](#)
- [Configuring locators, page 251](#)
- [Using JSP pages outside a component, page 257](#)

Component features

The Documentum component library contains a set of components that provide all of the common Documentum functions such as content transfer, inbox, folder browser, properties, and permissions.

Components have the following features:

- Composition

Components are composed of an XML definition that references JSP pages, a component behavior class, and an externalized strings resource file. Component layout and logic are defined in the following types of resource files:

- Layout: Layout is defined in JSP pages using HTML and configurable Documentum JSP tags..
- Logic: Component behavior is defined in a component class, and component is referenced by a logical name (component ID). .

- Independence from presentation technology

A component may be implemented using one of many presentation technologies such as JSP, XSLT, and WDK 5.x. This allows for the most appropriate user interface presentation technology to be chosen for the component or application. A component implementation may be migrated or updated from one presentation technology to another with minimal impact to the caller of the component, for example, JSP to WDK WDK 5. Context sensitivity may be applied to any component regardless of presentation technology employed.

- Context-sensitivity

Components can be configured to support context-based UI alternatives for each component. Context can be defined as the user role, the object type, or the object lifecycle state, for example. The component can have several different user interfaces and behaviors based on the context. The specific UI that is presented to the user is driven by the context parameters that are sent to the component, such as the object ID or user role. These component parameters are specified in the component definition XML file.

Context sensitivity allows for the extension of a component, for example, to support the introduction of new types, without affecting the caller.

- Reuse

Each component supports a contract public interface) through which all other components and containers (application or portal) communicate with the component. The contract consists of parameters for initializing the component, events for responding to interactions made by the user, and properties for interrogating the state of a component at any given time. The contract discourages access to the internals of a component, thus allowing the implementation to change over time without impact to the caller. Dependencies between components are broken, allowing individual components to be reused in multiple containers without the need to pull in dependent components.

- Configurability

Component exhibit configurable aspects of their user interface. Both the layout and behavior of the component may be declaratively modified without rebuilding the component. As the component is initially developed, the configuration points must be taken into account and built into the component. Generally, each configuration of a component is tied to a different calling context.

- Customizability

To achieve reuse, a component can be extended and customized through code development. This allows an extended component to provide additional or alternative behavior based on the calling context.

- Implementation

When components are addressed via URL, the URL points to a specific UI implementation (early binding). URLs for components can also be determined by context and dynamically generated at run time (late binding). WDK supports the following component implementation mechanisms: JSP, XSLT, or WDK 5. Select the appropriate UI implementation for your component. You can have components of several implementations in the same Web application.

The WDK framework uses a component dispatcher to call components. The component dispatcher maps each component URL to the appropriate implementation URL. With the exception of the login and changepassword components, each Documentum component requires a repository connection. If that connection is not available when the component is called, the component dispatcher opens the login component to create one. You can establish this connection silently by using a trusted connection.

- Loose coupling with component caller

In a Web-based application, the container (application or portal) issues a direct reference to a reusable page via a URL. The URL points to a specific user interface implementation, therefore restricting the configuration and extensibility of the application. In effect, the container and component are early bound, that is, tied together at development time. In order to support the WDK component requirements, the component definition infrastructure calls components indirectly. In effect, late binding is provided in which the user interface implementation is derived and the appropriate URL generated at runtime.

Component configuration file

The UI of a component is configurable via an XML configuration file. The file contains a component definition within the elements `<component></component>`. Both the layout and behavior of the component may be modified in the configuration file without rebuilding the component.

Each definition of a component is tied to a different calling context. The context is specified as the value of the scope element in the component definition by the configuration service. The user's context is used to look up the appropriate component definition.

Note: After you change XML configuration files, you must refresh the configuration definitions stored in memory. To refresh component definitions, navigate to the refresh-utility page *APP_HOME/wdk/refresh.jsp*, or restart the application server.

A sample component configuration file is displayed below. Strings in italics are customer-defined.

```

<config version="1.0">
1<scope qualifier_name="qualifier_value">
2<component id="component_name" version="version_number">
3<desc>Description goes here.</desc>
4<params>
  <param name="some_param" required="true">
    true>
  </param>
</params>

5<pages>
  <start>/custom/.../some_page.jsp</start>
  <custom_page_name>path_to_custom_page
  </custom_page_name>
</pages>

6<class>com.documentum.webcomponent.library.contenttransfer.importcontent.
ImportContent
</class>

7<service>
  <service-class>com.documentum.web.contentxfer.impl.ImportService
  </service-class>
  <transport-class>com.documentum.web.contentxfer.ucf.UcfContentTransport
  </transport-class>
  <processor-class>com.documentum.web.contentxfer.impl.ExportProcessor
  </processor-class>
</service>

8<init-controls>
  <control name="downloadDescCheckbox" type="
  com.documentum.web.form.control.Checkbox">
    <init-property>
      <property-name>value</property-name>
      <property-value>true</property-value>
    </init-property>
  </control>
</init-controls>

9<nlsbundle>com.documentum.webcomponent.library.
  attributes.AttributesNlsProp</nlsbundle>
10<custom_element>custom_element_value</custom_element>
11<helpcontextid>attributes</helpcontextid>
</component>

```

```
</scope>  
</config>
```

- 1 Defines the context in which the component definition is applied. This context is defined by a scope qualifier (attribute on the <scope> element) such as type, role, repository, location in the tree, or other qualifier that is defined for the application. To add a user-defined scope, you must create a qualifier class and declare it in the application app.xml file.
- 2 Defines the component. The component is identified by its id attribute. The optional component inheritance from a parent component definition is specified in the extends attribute. For more information on inheritance, refer to [Extending XML definitions, page 51](#). The component version support is described in [Versioning, page 55](#).
- 3 Optional element that describes the component. The description is displayed in the componentlist component, which displays information about each component in the application.
- 4 The params element contains the parameters used by the component behavior class. The <param> element names a parameter that is used by the component class. If the value for the attribute 'required' is true, the framework enforces the presence of an input value to the component. The component behavior class must implement code to use the value for each named parameter.



Caution: If the <params> element is empty, the configuration service will not check for the presence of parameters that are required in the parent definition. You must include all parameters defined in the parent definition as well as any new parameters that are used by your behavior class.

- 5 Contains all named component presentation pages. The component behavior class or start JSP page must implement code to call each named page. The <start> element value specifies the full path, relative to the Web application root directory, to the first component JSP page to be displayed. You can add custom elements whose name corresponds to the name of a custom page. The value of the custom element is the full path, relative to the Web application root directory, to the named component JSP page. The component is responsible for implementing navigation to the custom page. For information on implementing navigation within a component, refer to [Navigating within a component, page 437](#).
- 6 Contains the fully qualified class name for the component class.

- 7** `<service>` contains the service class and transport class elements, for content transfer containers, and the processor class, for content transfer components. For more information on these classes, refer to [Content transfer service classes, page 532](#).
- 8** Contains control initialization settings for controls in the component JSP pages. For more information on these classes, refer to [Content transfer control initialization, page 534](#).
- 9** Specifies the class that contains externalized strings for the component class and JSP pages. Properties files in the bundle can be localized. The file is located in the `/strings` directory under the application layer root directory.
- 10** One or more user-defined elements that are used by the component behavior class. The component must implement code to use the tag value.
- 11** Specifies an ID that is matched to the help component list of help topics. The referenced topic HTML page will be displayed in the help window when the user clicks a help button or link in the component UI.

An optional `<filter>` element can contain other elements such as `<component>` or a user-defined element. The filter element specifies a qualifier value for the filter. For example, the attributes component definition contains a filter element around the `<enableShowAll>`. The filter specifies the attribute role and value of administrator. This is matched to the user context, so that a user with the role of administrator is permitted to see all attributes of an object.

Component inheritance (extends)

You can extend a component to inherit the configuration of the parent component, override part of the configuration, or add to the definition. This inheritance is based on the `extends` attribute of the `<component>` element. There is no limit to the number of levels of inheritance. For example, a base properties component scoped on the `dm_sysobject` type can represent the default behavior of all type-scoped properties components. The properties component can then be extended to define a properties component scoped to a user-defined SOP type. The extended component inherits its definition from the base definition and overrides values or adds parameters specific to the new scope.

When you extend a component definition, define only the elements that override the base definition. All other elements are inherited. Make all of your modifications to a component definition in the `/custom` application directory. This ensures that when you subsequently migrate to a newer version of WDK or Webtop, your definition will inherit changes to the underlying component. Alternatively, if you do not want your

component to be subject to changes upon upgrade, copy from the base definition the elements that you do not want to change.

The strings for a component can be inherited from a component that you extend. Refer to [Inheriting strings, page 139](#) for details.

Example 5-1. Extending a component

A base properties component scoped on the `dm_sysobject` type can represent the default behavior of all type-scoped properties components. The properties component can then be extended to define a properties component scoped to a user-defined type. The extended component inherits its definition from the base definition and overrides values or adds parameters specific to the new scope.

The inheritance is specified as an `extends` attribute on the `<component>` element. In the following example, the definition extends the properties component definition for `dm_sysobjects`:

```
<scope type="SOP">
  <component name="properties"
    extends="properties:dm_sysobject_properties.xml">
```

The `extends` attribute has two parts:

- Name of the component in the parent component definition
- Name of the parent component definition

Continuing the above example, the SOP properties component can override the start page:

Base value in the `dm_sysobject` definition:

```
<pages>
  <start>/app/properties/properties.jsp</start>
  ...
```

New value in the SOP definition:

```
<pages>
  <start>/sop/properties/properties.jsp</start>
  ...
```

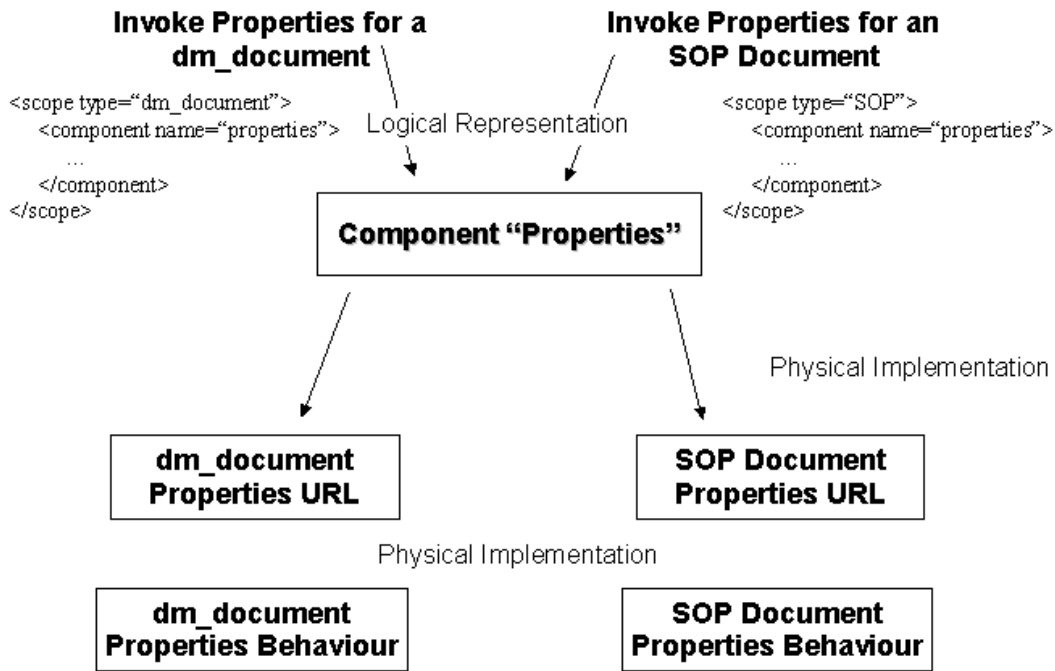
Component scope

You can limit the definition of a component to user contexts that match a particular qualifier value. For example, you can limit the definition based on type, role, doctype, or custom qualifier. This limitation based on qualifiers uses the `scope` attribute on the `<component>` definition.

The configuration service supports one or more scopes and one or more component definitions within each scope. For example, your properties component can have a component definition for `dm_document` and one for the custom SOP object type.

Note: Because component definitions can be extended, it is easier to maintain scalability in your application by using a separate configuration file for each scoped component definition. In this way, each definition can be maintained separately.

Figure 5-1. Scoped configuration



Scope based on object type is implemented by `com.documentum.web.formext.config.DocbaseTypeQualifier`. This class determines the type of the object by using the component parameter "type" or "objectId". Your component definition should have one of these two parameters as a required parameter. For example:

```
<config>
<scope type="dx_document">
<component id="dx">
  <params>
    <param name="objectId" required="true"></param>
  </params>
</component>
</scope>
</config>
```

Hiding components

You can hide component functionality for contexts that match a qualifier value using the scope attribute on the <component> element. You can also hide component functionality for a defined scope using the notdefined attribute.

If you scope a component definition to one qualifier value, the definition will apply only to contexts that match that qualifier value. For example, if you scope a properties definition to dm_folder, the definition will apply only to dm_folder and its subtypes but will not apply to dm_document objects.

You can also hide a component using the notdefined attribute of the <component> element. You would use the notdefined attribute to exclude a specific value of a qualifier. In the following example, the checkout component is not available for objects of type dm_folder:

```
<scope type='dm_folder'>
  <component id="checkout" notdefined="true"></component>
</scope>
```

In another example, when you display the properties page for folders, you want to display the permissions component only to administrators. You create two component definitions: one for the role of administrator, one for all roles. The scope type value in both definitions is dm_folder. The first definition below hides the properties component for folders for all users. The second definition makes the component visible for users with the role of administrator:

```
<scope type="dm_folder">
  <component id="properties" notdefined="true">
</scope>

<scope type="dm_folder" role="administrator">
  <component id="properties" extends="
    properties:webcomponent/library/properties/properties_component.xml">
    ...
</scope>
```

The decision whether to hide a component using scope or notdefined depends on how specific your exclusion must be. For example, if all qualifier values except one can use your definition, you can exclude the one value using notdefined. If you have a component that only applies to a specific qualifier value, you can hide the component from all others by limiting your definition to the specific value. In the following example, the checkout component is limited to dm_document objects. All other objects types (except subtypes of dm_document) will not have a defined checkout component and thus cannot be checked out:

```
<scope type="dm_document">
  <component id="checkout">
    ...
</scope>
```



Caution: You may find that other context values are not excluded by this component definition, if the component has been defined elsewhere in the application. For example, there is a checkout component in the WDK library that applies to `dm_sysobject` types. To exclude context values, you must add a definition that uses `notdefined`. In the following example, the checkout component is excluded for `dm_folder` objects:

```
<scope>
  <component id="checkout">
</scope>

<scope type="dm_folder">
  <component id="checkout" notdefined="true"></component>
  ...
</scope>
```

Hiding component features

Filter tags within configuration blocks allow parts of the definition to be shown or hidden based on the `<filter>` element. The filter is defined as the value of a qualifier attribute on the `<filter>` element. A filter can be set for any type of configuration service qualifier such as role, object type, or repository. The filter is applied to the user's context to evaluate whether the element is in force or not.

You can think of filters as an AND operation of qualifiers. The scope qualifier on the `<component>` element limits the application of the definition, and the scope qualifier on the `<filter>` element limits part of the definition even further.

Filters define the visibility of the contained elements. The filter scope name/value attribute pairs define the scope for the elements contained within the filter element. In the following example, the filter configures a tree component to display the admin node for the administrator role only:

```
<filter role="administrator">
  <node componentid="admin">
    ...
  </node>
</filter>
```

Filters can use the `clientenv` qualifier so that the contents of the filter element apply to a specific client environment such as a Web browser, portal application, or Application Connectors. The default value is `*`, meaning all client environments. This default is set in `/wdk/app.xml` as the value of `<environment>.<clientenv>`. This default is overridden in the `Webtop app.xml` to be `webbrowser` and a portlet application to be `portal`. Valid values are `webbrowser`, `appintg` (Application Connectors), `portal`, and the not operator, such as `not appintg`.

Filter tags can be placed anywhere below a primary element.

Configuring data columns

The following table shows the elements that are generally available in components that support configuration of data columns. For components that support preferences, the columns configuration elements in the component definition set the default view, before the user has selected column preferences.

<showfilters>	Set to true to show the objectfilters drop-down control
<objectfilters>	Contains filters that define which objects should be shown in the objects selection list.
<objectfilter>	Specifies a filter for the items that are displayed. Contains <label>, <showfolders>, <type>
<label>	Displays a label such as Folders or All. Can contain a string or <nlsid>.
<showfolders> <type>	To show folders only, set <showfolders> to true and <type> to null (no value). To show objects only, set <showfolders> to false and <type> to dm_sysobject. To show all, set <showfolders> to true and <type> to dm_sysobject. <type> can take any value that is a valid Documentum type.
<columns>. <loadinvisibleattribute>	Uncomment this element and set to true to get invisible attribute values for use in your component. The invisible attributes can then be passed by configuring a column in the <columns> element. Refer to Adding custom attributes to a datagrid, page 407 for details.
<columns><column>	Specifies columns to show or hide
<column><attribute>	<attribute> sets the attribute to be displayed in the column.
<attribute> <label>	The value of <label> sets a label for the column.
<column><attribute> <visible>	Set <visible> to true to show the column.

Adding or removing static data columns

Most common sysobject attributes are included as columns included in the WDK component definitions. You can change the column visibility to show or hide the defined columns. Insert new attribute columns that you require, such as r_object_id and

`object_name`. You can add a static column as a label, image, or `datasortlink` (making a sortable column). For example:

```
<dmf:columnpanel columnname='namePanel'>
  <th align='left' scope='col'>
    <dmf:datasortlink name='sortcol1' nlsid='MSG_NAME' column='object_name' .../>
  </th>
</dmf:columnpanel>
<dmf:columnpanel columnname='propbutton'>
  <th align='left' scope='col'>
    <dmf:image name='prop' nlsid='MSG_PROPERTIES' src='images/space.gif'/>
  </th>
</dmf:columnpanel>
```

If you need to display custom attributes, you must use dynamic columns. Refer to [Configuring dynamic data columns, page 232](#) for information.

To configure default columns for a component:

The steps to configure columns in a new component definition are the following:

1. Add settings for each column in the component definition. In the following example, the `alias` element is optional:

```
<columns>
  <column>
    <attribute>object_name</attribute>
    <alias>name</alias>
    <label><nlsid>MSG_NAME</nlsid></label>
    <visible>true</visible>
  </column>
  ...
</columns>
```

2. In your custom component class, import the utility class `com.documentum.web.formext.component.ComponentColumnDescriptorList`. This will automatically set the fields and label attribute on celltemplate controls on your component JSP page.
3. Process the configuration in your component class. The following call in a component class `readConfig()` method reads the list of visible columns:

```
private void readConfig()
{
  // read the list of visible columns
  m_columns = new ComponentColumnDescriptorList(this, "columns");

  // read and process other config element values
  // read the show folders boolean value
  Boolean bShowFolders = lookupBoolean("showfolders");
  if (bShowFolders != null)
  {
    m_fIncludedFolders = bShowFolders.booleanValue();
  }
}
```

The `ComponentColumnDescriptorList` object will read in the column configuration in the component definition and will then set the fields and labels attributes on

celltemplate controls on the component JSP page during rendering. For example, in the JSP page you have a column defined as follows:

```
<dmf:celltemplate type="date">
  <dmf:label/>:
    <dmf:datevalueformatter type="short">
      <dmf:label datafield="CURRENT"></dmf:label>
    </dmf:datevalueformatter>
</dmf:celltemplate>
```

All date attributes in the component definition that are specified as visible will be rendered. In the example below, two date columns will be rendered:

```
<columns>
  <column attribute="owner_name">singleselect</column>
  <column attribute="group_name">>false</column>
  <column attribute="r_creation_date">>true</column>
  <column attribute="r_modify_date">>true</column>
  <column attribute="r_access_date">>false</column>
  <column attribute="r_version_label">>true</column>
</columns>
```

The columns of data are rendered by `<dmf:datagridRow>`. If you need to add static rows between each row, close the table row generated by `datagridRow` using `</tr>`, then open an HTML table row and add your static row. The closing `datagridRow` tag will close your HTML table row tag.

Example 5-2. Adding a static row between each data row

The following example adds a space between each data row:

```
<dmf:datagridRow height='24'>
  <dmf:columnpanel columnname = 'name'>
    <td width=250>
      <dmf:label datafield='name' />
    </td>
  </dmf:columnpanel>
  <dmf:columnpanel columnname = 'description'>
    <td width='500'>
      <dmf:label datafield='description' />
    </td>
  </dmf:columnpanel>

  <!-- to add a separator row in the grid -->
  <!-- we must close the table row here and open another -->
</tr>

<!-- open the static row here -->
<tr height='1' class='doclistbodySeparator'>
  <td colspan='4' class='rowSeparator'>
    <img src='<%=Form.makeUrl(request,
      "/wdk/images/space.gif")%>' width='1' height='1'>
  </td>
<!-- close the datagrid row here -->
</dmf:datagridRow>
```

Configuring dynamic data columns

Instead of a fixed set of columns in a fixed order, you can define a list of template columns. A template defines the pattern for an unknown number of columns that are based on type, a specific attribute, or generic (any attribute). This allows you to create pages that do not specify which attributes are available or in which order they should be displayed.

The attributes and their order are resolved by the template system at run time. These unknown columns are formatted based on the type-based or generic templates.

To configure dynamic column display:

1. Add a `celllist` tag to a `datagridrow` element. Specify all of the possible data source field names as values for the `celllist` fields attribute. If one of the columns will display a custom attribute, set the `hascustomattr` attribute for the `celllist` tag to `true`. For example:

```
<dmf:celllist hascustomattr="true"
  fields='object_name,modifier,r_modify_date, r_creation_date, submitcode'>
```

Alternatively, you can open `<dmf:celllist>` and specify your fields in the `<dmf:celltemplate>` tag. If you do this, then the template will match only the fields specified in the `celltemplate` tag.

2. Add a `celltemplate` tag for each type of column that will be displayed. If more than one field matches the template, a column will be rendered for each match. In the following example, the first template is matched based on field name, so there is only one match. The second template matches by type, so two fields match and two columns are generated. The third template is generic, so it renders all remaining columns of data. The `dmf:label` tag displays the data dictionary label for the attribute.

```
<dmf:celltemplate field='object_name'>
  <td><dmf:label datafield='CURRENT' /></td>
</dmf:celltemplate>
```

```
<dmf:celltemplate type='date'>
  <td>
    <dmf:datavalueformatter type='short'>
      <dmf:label datafield='CURRENT' />
    </dmf:datavalueformatter></td>
</dmf:celltemplate>
```

```
<dmf:celltemplate>
  <td><dmf:label datafield='CURRENT' /></td>
</dmf:celltemplate>
```

Note: If your component definition specifies columns and labels, the label in the definition overrides the label in the JSP page.

Column order: The actual order of rendering is controlled by the order of fields in the fields attribute. The templates should be ordered as most specific first (field name), then semi-specific (type), and then generic.

Labels: The label datafield value of 'CURRENT' specifies that the label data value is taken from the current field. The actual value is determined dynamically.

HTML elements: HTML within a celllist tag is not rendered. The HTML must be within a celltemplate tag.

Sorting: You can place a datasortlink tag within a celltemplate without assigning a value to column, label, or datafield attributes. The following example makes a column sortable:

```
<tr>
  <dmf:celllist fields="object_name,title" labels="Document,Title">
    <dmf:celltemplate>
      <td align="left">
        <dmf:datasortlink name="sortcoll" datafield="CURRENT"/></td>
      </dmf:celltemplate>
    </dmf:celllist>
  </tr>
```

Component layout (JSP pages)

A JSP file contains layout for components. The layout is made up of HTML, JavaScript, and Documentum JSP tags.

The following topics describe the use of JSP pages in components:

- [JSP pages modeled by form class, page 233](#)
- [Contents of a WDK JSP page, page 234](#)
- [JSP includes, page 235](#)
- [Creating a component JSP page, page 236](#)
- [Using messages and labels, page 236](#)

JSP pages modeled by form class

The JSP page is modeled by a Form class that models a single web page, except in the case of included forms or containers. Each JSP page, unless it is an included page, has a <dmf:form> tag. The parent JSP page, which invokes form processing and the standard WDK JavaScript includes, has a <dmf:webform/> tag.

The Form class extends the Control class and implements event handler and navigation methods. Because the Component class extends Form, components inherit all of the behavior and methods of the Form class.

Forms are not configurable. Elements in the form are configurable, as JSP and HTML tags. Components that contain the form are configurable using a component definition.

WDK tags generate HTML 4.0 elements when the JSP page is rendered. Some tags generate a <table> element, such as the data grid control. The datagridrow tag generates a table row element (<tr>). To see how to use these tags within an HTML table, refer to the examples in the webcomponent library JSP pages.

Contents of a WDK JSP page

Following is a typical WDK JSP file:

```

1<%@ page contentType="text/html; charset=UTF-8" %>
2<%@ page errorPage="/wdk/errorhandler.jsp" %>
3<%@ taglib uri="/WEB-INF/tlds/dmform_1_0.tld" prefix="dmf" %>
4<%@ page import="com.documentum.webcomponent.environment.
  preferences.general.GeneralPreferences" %>
<html>
<head>
  5<dmf:webform/>
</head>
<body class='contentBackground'>
  6<dmf:form>
<table cellspacing=2 cellpadding=2 border=0 width="100%">
<tr>
  7<td>
    8<dmf:label cssclass='drilldownFileName'
      nlsid="MSG_CHOOSE_HEADER"/>
  </td>
  <td width='32' class='doclistHeader'>
    9<dmfx:docbaseicon size='32' name="object_icon"/>
  </td>
</tr>
10...
</table>
</dmf:form>
</body>
</html>

```

- 1** Sets the encoding of the page to a global character set (mandatory)
- 2** Specifies the error handling JSP (mandatory)
- 3** Specifies the tag libraries (mandatory if tags are used in the JSP). For example:

```
<%@ taglib uri="/WEB-INF/tlds/dmform_1_0.tld" prefix="dmf"%>
```

- 4 Imports the class that is referenced in a JSP scripting element on the page. For example:

```
<%@ page import="com.documentum.wp.app.WpTracing"%>
...
<dmf:text
  name="<%= WpTracing.DFC_TRACE_FILE_PATH %>" size="30"/>
```

- 5 Starts the form processor and generates the JavaScript includes to support client-side events. The tag must occur in every top-level form, before the HTML <body> tag. If a JSP is included in another JSP parent, it cannot have a <dmf:webform/> tag. Your component JSP page should not declare values for the formclass, nlsclass, or nlsbundle attributes on the <dmf:webform/> tag, because these values are defined in the component definition. The objects used for form class, nls class, and nls bundle can change dynamically depending on the component's context.

- 7 Prevent table cell contents from wrapping by adding a nowrap declaration, for example, <td nowrap>.

- 6 Generates the HTML form that is returned to the browser. The tag must occur between the <html> and <body> tags, after a <dmf:webform/>tag.

- 8 Specifies an HTML control that is defined in a Documentum tag library. The control tag generates HTML to the browser. The control can be configured to specify its appearance and data source. Controls can also specify an event handler, which is generally located in the component that includes the control in the JSP page. This allows the control event to be handled differently based on the component implementation. For example, an OK button in the import component behaves differently from an OK button in the delete component.

- 9 Specifies a repository-enabled control that is defined in a Documentum tag library.

- 10 Can be any valid HTML 4.0 or JavaScript.

JSP includes

JSP pages can include other pages using the JSP include directive <%@ include file=...>. The contents of the included file are merged before the JSP page is compiled. Subsequent requests to the including page do not detect changes to the included file.

Note: URLs in JSP pages must have paths relative to the Web application root context or relative to the current directory. For example, the included file <%@ include file='doclist_

thumbnail_body.jsp' %> is in the same directory as the including file. The included file <%@ include file='/webcomponent/navigation/drilldown/drilldown_body.jsp' %> is in the /webcomponent subdirectory of the Web application.

Creating a component JSP page

Create a layout that is appropriate for each of the contexts you have defined for your component. For example, the attributes component definition specifies three component definitions based on the context of object type: sysobject, folder, and document. Two different type-based layout start pages are defined: attributes_dm_folder.jsp and attributes_dm_document.jsp.

The top-level JSP page must contain a <dmf:webform> tag, which binds the JSP page to the WDK runtime, generates JavaScript and CSS file inclusions, and starts the form processor. Place the <dmf:webform/> tag before or within the HTML head elements of the JSP. All JSP pages must have a parent page that contains the <dmf:webform tag>.

Use the <dmf:form> tag in place of HTML <form> tags in all JSP pages that require the WDK framework. Place the <dmf:form> open tag directly after the HTML <body> tag in your JSP. Place the </dmf:form> close tag should be placed just before the closing HTML </body> tag. The remainder of the form contents should be placed within the <dmf:form> tag. All control events that are handled by the WDK runtime are submitted as HTML form POSTs.

Refer to [Component navigation, page 237](#) for information on navigation between pages in your component.

Add event handlers to your component class for events fired by the controls on the JSP page. In the following example, the event handler for a **Cancel** button returns to the calling component:

```
public void onCancelClicked(Control control, ArgumentList args)
{
    setComponentReturn();
}
```

Using messages and labels

Components have component-specific messages for the user interface: information messages, diagnostic error messages, labels, and window titles. For example, the newfolder component posts a message when a new folder is created.

The content of the message is contained in the NLS properties file for the component, and the message is retrieved as the nlsid attribute of a Documentum tag library element. Refer to [Working with XML configuration files, page 49](#) for more information about the

nlid attribute. Refer to [Configuring and localizing strings, page 137](#) for information on changing or adding strings to the UI.

For information on generating messages in your custom components, refer to [Rendering messages to users, page 600](#).

Using a raw JSP or static HTML file

Any technology that can render HTML can be used to render a component page. To support non-WDK JSP pages, HTML pages, or pages from other scripting languages, you must change the component definition. To use a raw JSP page or static HTML file, create a component definition XML file to specify your JSP or HTML page as the value of the `<pages>.<start>` element. Leave the component class value empty. For example:

```
<component name="info"
  <params>
    <param name="objectId" required="true"></param>
  </params>
  <pages>
    <start>/custom/info.html</start>
  </pages>
  <class></class>
</component>
```

There are a number of differences between this component definition XML file and one associated with a WDK 5 component:

- The static HTML page is named within the `<start>` element.
- There is no reference to a component class definition through a `<class>` tag.
- There is no reference to a resource bundle through a `<nlbundle>` tag.

Component navigation

Component navigation is described in the following topics:

- [Calling components by URL, page 238](#)
- [Calling components from an action \(LaunchComponent\), page 239](#)
- [Calling components from JavaScript, page 239](#)
- [Including a component in another component, page 240](#)
- [Navigating using browser history, page 241](#)

For information on component navigation using component APIs, refer to [Navigating within and between components, page 437](#).

Calling components by URL

Components are addressed by URLs. The component dispatcher maps the component URL to the appropriate page implementation URL.



Caution: If you use a URL in a JSP page, the URL must have its path relative to the Web application root context, not relative to the current directory.

The format of the URL that calls a component is the following:

```
/APP_HOME/component/component_name/  
[page_name][?params]
```

where:

- `APP_HOME`
Deployed application root context directory
- `component_name`
Name of the component, including container components, as defined in the component definition XML file
- `page_name`
Logical page name defined in the component definition XML file. If not present, the page defined by the `<start>` element is used.
- `params`
(Optional) Scope parameter and value pairs. If there are scoped definitions for the component, the parameters that you specify in the URL are used to dispatch the appropriate component definition.

Example 5-3. Calling a component by URL

In the following example, the `publish` component is called along with the object ID parameter that is required by the `publish` component:

```
http://localhost/wp/component/publish?objectId=xxx
```

Example 5-4. Calling a component in a container by URL

Some components must be called within a container. Call the container component as described above, and supply the contained component as a URL argument.

In the following example, the `attributes` component is called along with the object ID, so that the attributes for the object will be displayed. Because the `attributes` component is designed to be used within the `properties` container, the URL includes the container:

```
http://beauty.documentum.com/webtop/component/properties?  
component=attributes&objectId=0900000180309a58
```

Note: You cannot call a container by URL if the container extends the combocontainer, for example, the content transfer containers. The combocontainer is called by the LaunchComponent action execution class, which encodes and passes the required arguments to the container.

Calling components from an action (LaunchComponent)

Specify `com.documentum.web.formext.action.LaunchComponent` as the action execution for your custom action class if your action should launch a component. The LaunchComponent execution class invokes a component, with or without a container, on execution. The component element is required, and a container element is optional.

When an action invokes a component that is within a container, you must specify the container in your action definition. Use the combo container for multi-select actions. For example, the `dm_document_actions` definition specifies that the delete action launches the combocontainer.

Example 5-5. Launching a component from an action

The `newcabinet` action in WDK launches the `newcabinet` component in `newcabinetcontainer`. If the container is specified but no component is specified, then the default component for the container will be launched. If the LaunchComponent class is used but no component or container is specified, then the component dispatcher will launch a component with the same ID as the action.

```
<execution class="com.documentum.web.formext.action.LaunchComponent">
  <component>newcabinet</component>
  <container>newcabinetcontainer</container>
</execution>
```

When the LaunchComponent execution class is used, you can specify the type of navigation to the component that is to be launched. The navigation type is specified as the value of the `<execution>.<navigation>` element. Valid values: `jump` | `returnjump` | `nested`.

Calling components from JavaScript

Client-side functions post navigation server events. Use client-side navigation functions when you need to handle a client-side event by nesting or jumping to another component. The JavaScript file `/wdk/include/componentnavigation.js` contains the following client-side component navigation methods:

- `postComponentJumpEvent()`: Jumps to another component.

To perform the jump, the JavaScript function calls `postServerEvent()` with the server event "onComponentJump". For example, in Webtop the page `tabbar.jsp` contains a JavaScript that calls `postComponentJumpEvent()`. The component parameter that is supplied to `postComponentJumpEvent()` is the user's selected tab:

```
postComponentJumpEvent(null, component, "content",
    "processStartupAction", "true");
```

The arguments for this function are the following:

- `strFormId`: The target form for the event. If null, the first form on the page is assumed.
- `strComponent`: The target component URL for the jump or nest.
- `strTarget`: The target frame (optional). Default is the current frame. If target frame does not exist, a new window will pop up.
- `strEventArgName`: Event argument name (optional)
- `strEventArgValue`: Event argument value (optional)
- `postComponentNestEvent()`: Nests to another component with the same arguments as `postComponentJumpEvent`.

The following JavaScript function nests to the preferences component:

```
function onClickPreferences ()
{
    postComponentNestEvent(null, "preferences", "content", "component",
        "general_preferences");
}
```

Note: You must issue the jump or nest call from the enclosing container JSP page, if there is one. A reference to the JavaScript file `componentnavigation.js`, which contains the `postComponentXXXEvent` functions, is automatically generated with every HTML page.

To open a component in a new window, you can use the following JavaScript syntax:

```
function onClickopen ()
{
    newwindow = window.open("/" + getVirtualDir() + "
    /component/your_component", "your_component", "
    location=no,status=no,menubar=no,toolbar=no,resizable=
    yes,scrollbars=yes");
    newwindow.focus();
}
```

Including a component in another component

There are two ways to include a component within another component. The first is to use a component container, which is documented in [Configuring containers, page 243](#).

Containers provide support for paging, change queries and notification, and NLS text strings.

The second way to include components is using the `componentinclude` tag from the Documentum tag library. The included component is rendered in place of the `componentinclude` tag. The `componentinclude` tag has the following syntax. The component name must match the name of the component in its XML configuration file:

```
<dmfx:componentinclude name="instance_component_name"
  component="component_name"></dmfx:componentinclude>
```

For example, the startworkflow component JSP page `startWorkflow.jsp` includes the `taskheader` component:

```
<dmfx:componentinclude component='taskheader' name='taskheader' page='startwf' />
```

You can pass arguments to the included component using an `argument` tag. In the following example, the `details` component includes an `object details` component that is updated from the `r_object_id` datafield:

```
<dmfx:componentinclude name="details" component="object_details">
  <dmf: argument name="objectId" value="418">
</dmfx:componentinclude>
```

Note: The included component JSP page cannot have the following tags: `<html>`, `<head>`, and `<body>`. These tags are provided by the host component page. If you set the `componentinclude` tag `"visible"` attribute value to `false`, the included component will not be rendered.

The `ContainerIncludeTag` class provides methods for getting the contained component ID, name, event arguments, and the current page of the component. The `containerinclude` class creates the contained component.

Navigating using browser history

The Form processor and Form class provide support for access to and control of browser history. Navigation history is modeled as a `FormHistory` object that contains a series of `FormHistorySnapshot` objects. The form history object represents a client frame or window. The snapshot represents the state of a page in the client frame or window.

The Form processor adds the current page to the `FormHistory` object as a `FormHistorySnapshot` (equivalent to state for a given form URL). The request number portion of the form request ID is used to retrieve a `FormHistorySnapshot` object. If the snapshot refers to a closed form, the form is reopened.

If a URL addresses a page whose history has been released because the number of history pages were exceeded, the URL is forwarded to `/wdk/historyReleased.jsp`.

The Form processor identifies and gets the FormHistory object using the FormRequestId that is passed on every HTTP request. If no form request ID was passed in the request, a new FormHistory object is created. If a URL addresses a page that has no request ID, the session is assumed to have timed out and the URL is forwarded to /wdk/timeout.jsp.

The FormRequestId object is created by the Form processor as a formatted string with two fields:

- client ID identifies the client browser frame or window. If a request is passed without a FormRequestId, a new client ID is generated in the series _client1, _client2, clientN.
- request number identifies the request index within the frame or window specified by the client ID. The index is consecutive and begins with 1. You can pass a FormRequestId with no request number, which will be interpreted as the last request for the given client ID.

Component operations on foreign objects

Operations on an object may be performed in the current repository, such as delete, or in the source repository, such as checkout or checkin. If the user is attempting an operation on an object that is in another repository such as an object in a multirepository search result set, virtual document with foreign attachments, or workflow task with foreign attachments, you can specify that the operation should be performed on the source repository.

If the component performs a query that must be executed against the source repository, you can configure your component to do this. To change to the object's repository, add a <setrepositoryfromobjectid> element to your component definition and set the value to true:

```
<setrepositoryfromobjectid>true</setrepositoryfromobjectid>
```

If the component does not execute a query, then you do not need to add this element to your definition.

Presubmission client events

Control events are handled either in the client, using JavaScript, or in the component class, using an event handler method. Control event handling is described in [Control events, page 165](#). If your component needs to do client-side processing before a JSP form is submitted, you can use the presubmission processing mechanism. For example, if you need to update hidden fields on the form after the user clicks **OK**, you need to perform presubmission processing.

Presubmission processing has two parts:

- Client
Register your client-side event handler by calling a JavaScript function, and provide the JavaScript event handler.
- Server
Register the event in your component class. This signals the framework to fire the client-side event before posting the server-side form submission.

To register the client-side presubmission handler and event:

1. Register the handler in your JSP page by calling `registerPreSubmitClientEventHandler(strSourceFormName, strEventName, fnEventHandler)`. This JavaScript function is available in all JSP pages. If the source form name is null, the event is handled regardless of the source form. For example:

```
<script>
    registerPreSubmitClientEventHandler(null, "invokeSubmit", onInvokeSubmit);
</script>
```

2. Place your JavaScript event handler in the component JSP page. For example:

```
function onInvokeSubmit(arg)
{
    //client-side processing here
}
```

3. In your component or control class render method, call `setPreSubmitClientEvent(String strClientEventName, ArgumentList clientEventArgs)` in your component class or within a custom control class. For example:

```
protected void renderEnd(JspWriter out) throws IOException
{
    ...
    ArgumentList clientArgs = new ArgumentList();
    getForm().setPreSubmitClientEvent("invokeSubmit", clientArgs);
}
```

Configuring containers

Many components share common UI and behavior or state. For example, dialogs have a title, content area, and a button panel. Containers provide common layout and behavior for multiple components. Components can be used within more than one container, inheriting their UI and behavior from the container.

If you do not need container layout and behavior, you can simply include one component within another using the `<componentinclude>` tag.

A container is a specialized component with the following characteristics:

- The container component definition includes a "component" parameter.
- The container class extends `com.documentum.web.formext.component`.
- The JSP page for the container has one `<containerinclude>` tag (multiple `containerinclude` tags are not supported).
- Control values that are changed by the user are propagated to all instances of the control in other embedded components. For example, a user selects two checked out documents and then selects Checkin. The user enters a description for the first file and selects Next. The description is propagated to the description field of the next file.
- A container can display only one component at a time. The currently displayed component is accessible through methods on the Component class.
- The components that are included in the container are defined in the `<contains>` element. The first component in a list of contained components is the default component.
- Containers have all of the Control, Form, and Component methods available to them, because they extend the Component class. Additional Container class methods support change query and notification, wizard navigation, and getting and setting contained components.
- The container start page is defined in the container configuration file as the value of the `<pages>.<start>` element.

The following topics describe containers:

- [Container types, page 244](#)
- [Calling containers, page 247](#)
- [Configuring containers, page 249](#)
- [Components that must run within a container, page 250](#)
- [Creating modal containers, page 251](#)

Container types

Use or extend a WDK container that is appropriate for your navigation purposes:

Table 5-1. Types of WDK containers

Condition	Use
One component Buttons OK, Cancel, Close, Help	dialogcontainer
One navigation component breadcrumb control	navigationcontainer

Condition	Use
One component with ordered pages (Previous , Next)	wizardcontainer
Multiple selection	combocontainer
Multiple contained components	propertysheetcontainer
Multiple contained components with wizard navigation	propertysheetwizardcontainer
Content transfer	contentxfercontainer

Abstract container — The generic container is an abstract container that has no visible layout. It does not support change notifications or wizard navigation. The generic container surrounds a single contained component with HTML <head> and <body> tags. When you invoke the container, you must supply the name of the contained component as a parameter.

The generic container is extended by the dialogcontainer.

Dialog container — The dialog container extends the abstract container and adds dialog support for a single contained component. The dialog layout contains a header and footer. The header includes a title (MSG_TITLE) and object label (MSG_OBJECT). The footer contains **OK** (MSG_OK), **Cancel** (MSG_CANCEL) or **Close** (MSG_CLOSE), and Help buttons. Change notifications are called as appropriate for each button event.

The OK and Cancel buttons are disabled if canCommitChanges() or canCancelChanges() return false, respectively. If both methods return false, the Close button is displayed.

Some examples of actions that use the dialogcontainer include removeuserorgroup, add_translation (Web Publisher), newchangeset (Web Publisher).

Navigation container — The navigation container wraps navigation components and provides a header with active breadcrumb and title controls. The breadcrumb allows the user to navigate to an object that is displayed, and the drilldown component is updated by the breadcrumb selection. Override this component definition to jump to another navigation component using a breadcrumb.

The navigationcontainer definition includes a <navigation> element that specifies the type of navigation.

Some examples of actions that use the navigationcontainer include versions, locations, relationships, renditions.

Wizard container — The wizard container extends the dialog container and adds wizard navigation support for a single contained component that has multiple pages. The wizard layout adds **Next** (MSG_NEXT) and **Previous** (MSG_PREVIOUS) buttons.

Component.hasNextPage() and hasPrevPage() are called on the contained component to determine when to enable and display the next and previous page navigation buttons. The methods onNextPage() and onPrevPage() are called to switch the contained component page when the user clicks the **Next** or **Previous** button. The component within the container must support onNextPage() and onPrevPage(), which returns false in the base Component class.

The taskcomponentcontainer in /webcomponent/config/library/workflow/taskmanager extends the wizardcontainer. Use this component as an example of how to extend the wizardcontainer. For information on programmatically navigating between next or previous components, refer to [Navigating within a container, page 441](#).

Combo container — The combo container extends the wizard container and adds support for multiple instances of the same contained component. This allows the user to perform multi-select operations for a component. An Apply to All prompt is displayed when the user selects multiple operations. Change notifications and paging methods are called as appropriate for each button event.

Control values that are changed by the user are propagated to all instances of the control in other embedded components. For example, a user selects two checked out documents and then selects Checkin. When the user enters a description for the first file and selects **Next**, the description is propagated to the description field of the next file.

Some examples of actions that use the combocontainer include delete, rename, sendto, removeattachment, abort, halt, or resume workflow.

Property sheet container — The property sheet container extends the wizard container and adds support for multiple contained components within a property sheet. The property sheet layout adds a tab (streamline view) or link (list view) for each contained component. The labels for the tags are defined as the value of the MSG_TITLE key in the properties resource file (*Prop.properties file) for each component.

The contained components are specified in the container XML configuration file, within the <contains> tag. The default component, which is displayed first, is the first component in the list. You can nest property sheets, so that the calling component is not closed. You would call onComponentNested() in your event handler.

Some examples of containers that extend propertysheetcontainer include preferences, locatorcontainer, properties, propertysheetwizardcontainer, advsearchcontainer.

Property sheet wizard container — The property sheet wizard container extends the propertysheetcontainer container and adds support for wizard-type navigation through the pages of a component (**Previous** and **Next** buttons) and through components in the container (tabs).

Some examples of containers that extend the propertysheetwizardcontainer include adminpropertysheetcontainer, newcabinet, newdoc, and newfolder containers.

Content transfer containers — Content transfer components must run within a content transfer container to support the following common processes:

- Getting configuration information
- Handling checking of applets installation on the client
- Getting lists of object IDs for operations on multiple files

The `ContentTransferContainer` base class supplies support for the common processing listed above, but it has no UI such as **OK**, **Next**, or **Previous** buttons. The container class gets multiple selection arguments and initializes a list of object IDs. The container looks up content transfer configuration settings from `app.xml` and servlet paths from `web.xml`.

The content transfer base container extends the combo container and implements `IContentXferServiceMgr`. The base content transfer container class provides methods to start the service, check applet installation, and get each of the configuration settings.

Calling containers

Containers can be invoked in the following ways:

- [Calling a container by URL, page 247](#)
- [Calling a container by JavaScript, page 247](#)
- [Calling a container from an action, page 248](#)

Calling a container by URL

You can call a component within a container from a JSP page or with a URL in the browser. When you call a container, the component parameter is a required parameter. The container must be invoked with a component name. Additional parameters, optional or required, may be specified in the container component configuration file. The contained component's arguments are passed as URL arguments. For example:

```
http://wt/component/dialogcontainer?component=checkin&objectId=objectid
```

Note: You cannot call content transfer containers, or any container that extends `combocontainer`, by URL. These containers are called by the `LaunchComponent` action execution class, which encodes and passes in the required parameters.

Calling a container by JavaScript

You can call a contained component from a JavaScript function or event handler in a JSP page. Two JavaScript functions post a server component page event to

a given URL: `postComponentJumpEvent()` to jump to another component, or `postComponentNestEvent()` to nest another component. These JavaScript functions are contained in the `/wdk/include/componentnavigation.js` file. The functions have the following parameters:

- `strFormId`: The target form for the event. If null, the first form on the page is assumed.
- `strComponent`: The target component URL for the jump or nest.
- `strTarget`: The target frame (optional). Default is the current frame. If target frame does not exist, a new window will pop up.
- `strEventArgName`: Event argument name (optional)
- `strEventArgValue`: Event argument value (optional)

For example:

```
postComponentJumpEvent(null, "search", "content", "queryType", "string",
    "query", strValue);
```

Calling a container from an action

You can specify in an action configuration file that a container and component will be launched for a given action. If the contained component is not named, the default component will be displayed within the container.

The action execution class in an action definition must be `LaunchComponent`, which will launch the container and pass in required parameters. If your action needs to check permissions on an object, use the execution class `LaunchComponentWithPermitCheck`. For example, content transfer and delete components need a permission check.

Specify the component to be launched and the container for the component as child elements of the execution element in the action configuration file. In the following example from `dm_sysobject_actions.xml`, the attributes `action` launches the history component within the properties container:

```
<action id="attributes">
  <params>
    <param name="objectId"></param>
  </params>
  ...
  <execution class="com.documentum.web.formext.action.LaunchComponent">
    <component>history</component>
    <container>properties</container>
  </execution>
</action>
```

You can also specify the type of navigation to the component: `jump`, `returnjump`, or `nested`. By default, the launched component is nested within the current component.

Configuring containers

Containers are configured with XML definitions in the same way that components are configured. The container definition has a `<contains>` element, with child `<component>` elements that are supported by the container. You can configure containers to require a visit to one or more components before changes are committed. You can also change container labels using the string properties file for the container. You can set the default component by listing it first within the `<contains>` element.

Require visit

The `requiresVisit` attribute on a component element requires a component to be visited before a change is committed, such as an OK button. The `requiresVisit` attribute can be set on any component in a container definition for containers that extend `propertySheetcontainer`. Set `requiresVisit` to `true` to declare that a particular component must be visited before the container can commit changes. In the following example, the attributes component must be viewed before the user can commit changes:

```
<contains>
  <component>newFolder</component>
  <component requiresVisit='true'>attributes</component>
  <component>permissions</component>
</contains>
```

Example 5-6. Requiring a component to be visited:

In the following example, the OK button will not be enabled until the attributes component has been visited:

```
<contains>
  <component>newFolder</component>
  <component requiresVisit='true'>attributes</component>
  <component>permissions</component>
</contains>
```

A component can also require a visit in its component definition, which will have the same effect as the component attribute setting in the container definition. The component must be contained within a container that extends `propertySheetcontainer`. The same example above would be as follows in the attributes configuration file:

```
<requiresVisitBeforeCommit>true</requiresVisitBeforeCommit>
```

Container labels

Documentum containers provide container labels. For example, the `dialogcontainer` contains a dialog header, OK, and Cancel buttons.

Contained components can override the NLS strings that are defined in a container's layout. The contained component must provide label values in an NLS properties file for the following NLS IDs in order to override the values for the container:

- MSG_TITLE: Container's title string, such as 'Properties'
- MSG_OBJECT: Container's object/subject string, such as 'foo.xml'
- MSG_PREV: Label for the Container button that navigates to the previous page
- MSG_NEXT: Label for the Container button that navigates to the next page
- MSG_OK: Label for the Container OK button
- MSG_CANCEL: Label for the Container Cancel button
- MSG_CLOSE: Label for the Container Close button

The contained component can also override a string by overriding the `Container.getString()` method.

Example 5-7. Overriding a string

The following example overrides `Container.getString` to set a string at run time:

```
public String getString(String stringId)
{
    String strResult = "";
    if ( stringId.equals("MSG_OBJECT" )
        {
            strResult = ...
        }
    else
    {
        strResult = super.getString(stringId);
    }
    return strResult;
}
```

Components that must run within a container

Some components cannot stand alone and must run within a container. The container supports multiple selection, calling the contained component for each object that has been selected.

The content transfer components (cancelcheckout, checkin, checkout, edit, export, import, and view) need access to content transfer configuration that is read by the container class. The newdocument, newfolder, and newcabinet components must also be used within their respective containers.

Creating modal containers

Any component can be modal. (Refer to [Using modal windows](#), page 425 for information on modal windows.)

By default, all nested components, including components within containers, are modal, and all other component navigation is not modal. Nested components are set to modal by the WDK framework. If you do not want your nested component to be modal, call `setModal(false)` in your component `onInit()` method.

You can use WDK template containers to develop your own modal container:

- Modal container with tabs
Use `webcomponent/library/properties/properties.jsp`.
- Modal container without tabs, for screens with a breadcrumb or no navigational element
Use `webcomponent/library/create/newContainer.jsp`
- Modal container with a datagrid in the content area
Use `webcomponent/library/async/jobstatus.jsp`. (Wrap datagrids with a 1-pixel box.)

Configuring locators

A locator presents the user with a UI to locate an object in a doctype. The type of the object can be defined via component configuration or passed as a parameter by the caller component.

The following locators are available for specific object types:

- Sysobject (generic) locators
 - `objectlocator`: Abstract base locator for a repository object
 - `sysobjectlocator`: Locates `dm_sysobjects`
 - `myobjectlocator`: Locates objects checked out and recently modified by user
 - `recentsysobjectlocator`: Locates `dm_sysobjects` recently used or selected (in locator) by user
 - `subscriptionlocator`: Locates `dm_sysobjects` subscribed to by user
 - `locatorcontainer`: Contains generic locators, allowing multiple located objects
- User and group locators (the locators treat the group as a logical subtype of the user)
 - `userorgrouplocator`: Locates users or groups

- recentuserorgrouplocator: Locates users or groups recently used or selected (in locators) by user
- grouplocator: Displays a hierarchal view of all dm_group objects
- recentgrouplocator: Locates groups that were recently selected or used in a locator
- grouplocatorcontainer: Contains group locators
- useronlylocator: Displays a hierarchal view of all dm_users
- recentuseronlylocator: Locates users that were recently selected or used in a locator
- useronlylocatorcontainer: Contains dm_user locators
- dm_document locators
 - alldocumentlocator: Displays a hierarchal view of all dm_document objects
 - mydocumentlocator: Locates documents checked out and recently modified by user
 - recentdocumentlocator: Locates documents that were recently selected or used in a locator
 - documentsubscriptionlocator: Locates dm_document objects subscribed to by user
 - documentlocatorcontainer: Contains dm_document locators
- dm_folder locators
 - allfolderlocator: Displays a hierarchal view of all dm_folder objects
 - recentfolderlocator: Locates folders that were recently selected or used in a locator
 - foldersubscriptionlocator: Locates dm_folders subscribed to by user
 - folderlocatorcontainer: Contains dm_folder locators
- dm_policy locators
 - alllifecyclelocator: Displays a flat view of all lifecycle objects
 - lifecyclefolderlocator: Displays a hierarchal view of all lifecycle objects
 - recentlifecyclelocator: Locates lifecycles that were recently selected or used in a locator
 - lifecyclesubscriptionlocator: Locates lifecycles subscribed to by user
 - lifecyclelocatorcontainer: Contains lifecycle locators
- dm_process locators
 - allwftemplatelocator: Displays a flat view of all workflow template objects

- wftemplatefolderlocator: Displays a hierarchal view of workflow template objects
- wftemplatesubscriptionlocator: Locates subscribed workflow templates
- mywftemplatelocator: Locates templates checked-out and recently modified by user
- recentwftemplatelocator: Locates workflow templates recently used by user
- wfemplatelocatorcontainer: : Contains workflow template locators

A locator can present different UIs depending whether multi-selection or single selection is enabled. Each locator component can be either run in a locator container or standalone. The locators within the same container work together to give a user a different view of the available objects. The locator configuration contains a <views> element that defines the following views:

- root

Displays a hierarchical list of root containers of the selectable objects, for example, cabinets or folders

Figure 5-2. Root (cabinet) locator

The screenshot shows a web application interface for a user named 'wendy2'. It features a search bar with a 'Go' button. Below the search bar is a table displaying a list of objects. The table has three columns: 'Name', 'Version', and 'Owner'. The objects listed are:

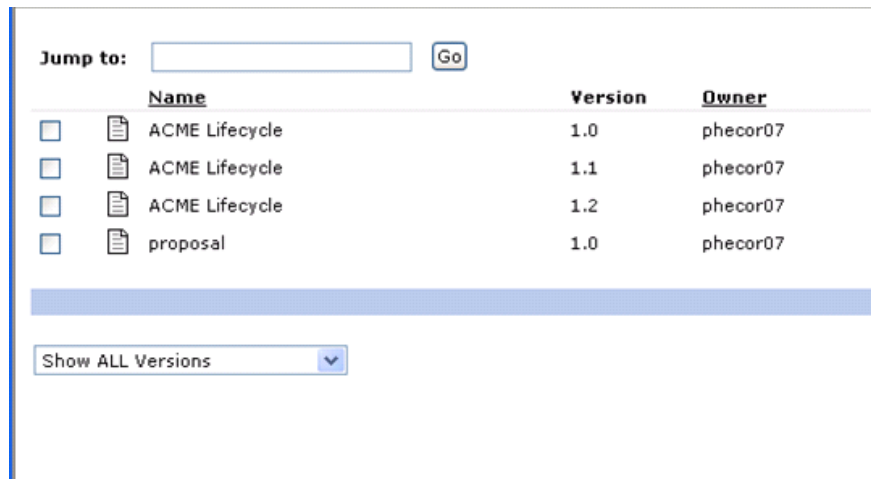
Name	Version	Owner
1) Equity Office		wendy2
101.		wendy2
a new cabinet for dave		Robert Black
a tab		wendy2
AA nicks test cabinet		wendy2
AA_sub123		wendy2
aaaaaa		wendy2
aaaaaaaaaaaaa nick		Robert Black
ABC		wendy2
ACME		wendy2

At the bottom of the table, there is a pagination bar that reads 'Displaying 1 to 10 of 81' followed by navigation icons and '1 of 9'.

- flatlist

Displays all the selectable objects that meet the criteria. By default, myobject and recently used object locators display results in a flatlist.

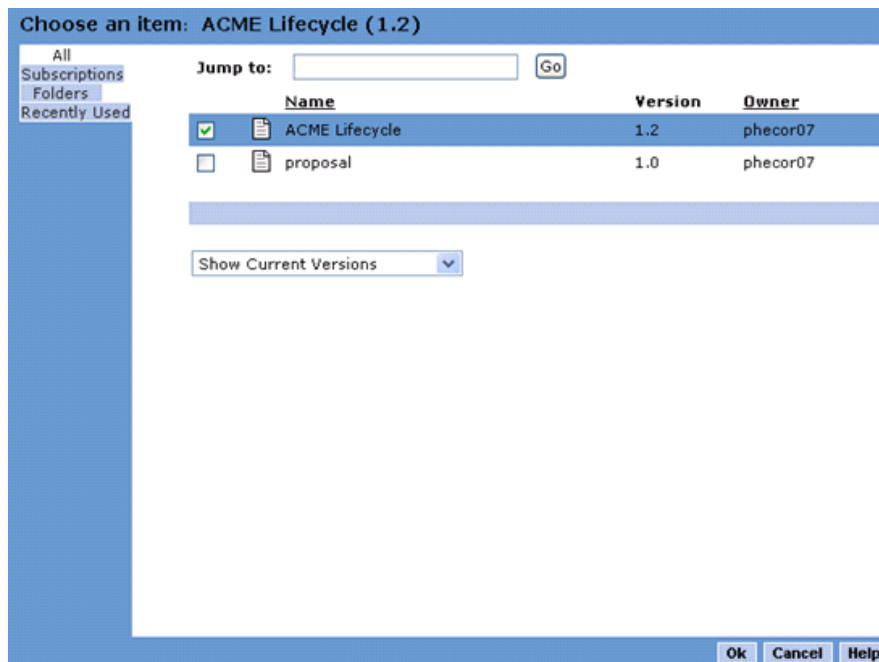
Figure 5-3. Flatlist locator



- container

Displays a hierarchical list of objects within the selected root container. By default, subscriptions locators display results in a container (hierarchical) list.

Figure 5-4. Container locator



For example, a `dm_sysobject` locator can define the cabinet list as the root view and `dm_folder` objects as the containers. You can configure whether the container is

selectable. If it is not selectable, the user will be allowed to drill down into the container, but the container itself can not be selected.

Filters provide queries to locate the objects. The query is built from the configuration file elements content in the following way, with configuration elements within square brackets ([element_name]):

```
select [objecttype] from [includetypes] where not type [exclusiontype] and
    [attribute1] [predicate1] [value1][and | or] [attribute2] [predicate2] [value2] ...
```

<objecttype>	Base type to be located
<containerselectable>	Set to true to specify that the container type is selectable when the container is a subtype of the objecttype.
<privatecabinetvisible>	(In sysobject locator definitions) Set to true to display in root view the private cabinets not owned by the session user
<allversionsvisible>	(In sysobject locator definitions) Set to true to display all versions of located objects
<privategroupvisible>	(In user or group locators) Set to true to display private groups
<flatlist>	Set to true to display a flat list of all selectable objects. Overrides the views available in the parent type.
<views>	Contains <view> elements
<view>	View element configures root (cabinet view), container, and flatlist views. The applyto attribute must specify one or more of the views in a comma-separated list, for example, applyto='root,container,flatlist'.
<queryfiltersets>	Contains <queryfilterset> elements that present a drop-down list that will be visible if there is more than one filter (<queryfilterset>) defined.
<queryfilterset>	Contains a set of queries contained in <queryfilter> elements that filter the selection list, for example, folders only. Defines one dropdown item. Each view can contain up to three filter sets.

<code><queryfilter></code>	Each filter contains a DQL query: select <code><includetypes></code> from <code><containertypes></code> not <code><excludetypes></code> where <code><attribute></code> <code><predicate></code> <code><value></code> <code><and></code> <code><attribute></code> <code><predicate></code> <code><value></code> ...
<code><queryfilter>.<displayname></code>	Specifies the name to be displayed for the queryfilter. Can contain a <code><nlsid></code> element or text string.
<code><queryfilter>.<containertypes></code>	Comma-separated list of navigable object types, such as <code>dm_cabinet</code> , <code>dm_folder</code> .
<code><includetypes></code>	Optional element (cannot be more than one instance of this element) that specifies a comma-separated list of Documentum types to be included in the view. If the container is not listed along with the subtypes within the container, it will not be included in the query
<code><excludetypes></code>	Optional element (cannot be more than one instance of this element) that specifies the types of objects to be excluded from the view.If a container type is not one of the subtypes listed in this tag, the containers are not excluded. If a container type is listed, the objects of the container type will be hidden, for example, <code>dm_folder</code> , <code>dm_document</code> hides all folders or documents.
<code><attributefilters></code>	Cannot be more than one instance of this element. Contains <code><attributefilter></code> sets that filter objects based on their attributes
<code><attributefilter></code>	Contains <code><and></code> , <code><attribute></code> , <code><predicate></code> , and <code><value></code> to compose an attribute filter
<code><attribute></code>	(Required) String attribute name, for example, <code>a_content_type</code> (single tag only)
<code><predicate></code>	Contains a logical operation such as equals. Valid values: <code>sw</code> (starts with), <code>ew</code> (ends with), <code>co</code> (contains), <code>nc</code> (not contains), <code>eq</code> (equal), <code>ne</code> (not equal), <code>gt</code> (greater than), <code>ge</code> (greater than or equal), <code>lt</code> (less than), <code>le</code> (less than or equal)

<code><value></code>	Use the attribute <code>dqlformatted='false'</code> to quote and escape a value. Use the attribute <code>casesensitive='true'</code> to require a case-sensitive comparison (must be true for integer attributes on Content Server/DB2 environment)
<code><and></code>	Boolean: true to combine attribute filters, false to perform OR filter (single tag only)

Using JSP pages outside a component

You can call JSP pages directly in your WDK 5 application if the pages do not require a session. If the JSP page requires a session, you must make it into a component. Your component definition for a JSP page must have a `<pages><start>` element that points to your JSP page. Do not specify a `<class>` element for your component.

The following example shows a configuration file for a simple component that uses a JSP page and requires a session:

```
<config>
<scope type="dm_sysobject">
<component name="sample">
  <pages>
    <start>/custom/sample/sample.jsp
  </start>
  </pages>
</component>
</scope>
</config>
```


Configuring Application Connector menus, components, and actions

These topics are included:

- [Overview, page 259](#)
- [Modifying the Documentum menu, page 259](#)
- [Customizing application connector components and actions, page 265](#)

Overview

EMC | Documentum Application Connectors enable Windows applications, such as Microsoft Word, Excel, and Powerpoint, to check in and check out files from Documentum repositories (as well as many other tasks, such as searching repositories) through a combination of .NET client assemblies and runtime services, and components running on EMC | Documentum WDK-based application server applications (for example, Webtop).

Modifying the Documentum menu

These topics are included:

- [Overview, page 260](#)
- [Removing menu items from all application connectors, page 261](#)
- [Modifying menu items for all applications, page 261](#)
- [Adding custom menu items to all applications, page 264](#)
- [Restricting menu items to specific applications, page 264](#)

Overview

EMC | Documentum Application Connectors create a standard Documentum menu in Microsoft Word, Excel, and Powerpoint. The standard Documentum menu items perform functions to access objects in repositories. You can remove and modify standard Documentum menu items as well as add your own custom menu items. Standard menu items and their functions include the following:

- **Login As** — Log in to a repository
- **Log out** — Log out of a repository
- **Create New from Template** — Create a new file from a template stored in a repository
- **Save** — Save a file to a repository
- **Save As** — Save a file with a different name or to a different location
- **Open** — Open an object from a repository
- Save a copy of a file into a repository
- **Search Repositories** — Search a repository
- **Properties** — Display the attributes of an object
- **Locations** — Display the locations of an object in a repository
- **Versions** — Display all the versions of an object in a repository
- **Renditions** — Display all the renditions of an object in a repository
- **Lifecycle > Promote** — Moves a file to the next state in a lifecycle.
- **Lifecycle > Demote** — Moves a file back to the previous state in a lifecycle.
- **Lifecycle > Attach** — Adds a file to the initial state in a lifecycle.
- **Lifecycle > Suspend** — Pauses a file in a lifecycle by moving it to a Suspend state.
- **Lifecycle > Detach** — Removes a file from a lifecycle.
- **Lifecycle > Resume** — Restarts a file in a lifecycle by moving it from the Suspend state to the state it was in before it was suspended.
- **Cancel Checkout** — Cancel checkout of an object (unlock)
- **Send to > Quick Flow** — Send file to a distribution list
- **Send > Email Recipient as Web Link** — Send file to single user (using a locator)
- **Send to > Workflow** — Start a workflow and attach the file
- **Help** — Display online help for your application connector.

The Documentum menu is constructed from a global menu that is configured in two sources: your application connector's `app.config` file and the application connector menu definition file, `appintgmenubar_menusgroup.xml` file, which is located in `/webcomponent/config/library/appintgmenubar`. At runtime, the `appintgmenubar_menusgroup.xml` file is downloaded to the local machine. The

application connector is initialized and is updated on demand—that is, the `appintgmenubar_menugroup.xml` file is downloaded whenever its version changes.

Note: The global menu for Application Connectors for Microsoft Office applications are located in `%PROGRAMFILES%\Microsoft Office\OFFICE{10, 11}`, where the variable `%PROGRAMFILES%` is the path to the Program Files directory on the client machine. This part of the application menu cannot be configured.

When a user chooses a menu item, a native modal dialog window that hosts a browser control is launched, and the corresponding WDK component or action specified in the `appintgmenubar_menugroup.xml` file is called. The browser control loads an action URL for the selected menu item. This URL consists of action ID, clientenv qualifer, theme, and repository.

Webtop uploads an action map each time a repository document is opened. The action map is based on the user session and the selected object. This action map enables or disables menu items based on the user context. For example, when the user logs into a new repository, a new action map is downloaded to the client. This dynamic menu structure is stored in client memory and is not persisted on the client.

Removing menu items from all application connectors

To remove a menu item from all application connectors:

1. On the WDK application server machine, copy the `/webcomponent/config/library/appintgmenubar/appintgmenubar_menugroup.xml` file to your application `custom/config` directory.
2. Extend the WDK version of this component by changing the `<menugroup>` element as follows:

```
<menugroup id="appintgmenubar" extends="
  appintgmenubar:webcomponent/config/library/appintgmenubar/apptintgmenubarmenugroup
```

3. In your custom version of `appintgmenubar_menugroup.xml`, delete the `<actionmenuitem>` element.
You can identify the menu item by finding the string in `<menuitem>.<value>` that matches the menu item name displayed on the menu.
4. Save the `appintgmenubar_menugroup.xml` file and restart the application server.

Modifying menu items for all applications

To modify menu items that call application connector components and actions in Webtop, you modify elements and attributes in a `<menugroup>` element in

/webcomponent/config/library/appintgmenubar/appintgmenubar_menugroup.xml.
 Refer to [Table 6-1, page 262](#) for a description of the appintgmenubar_menugroup.xml file elements.

Note: Each action or component in the menu definition (that is, the value of the <action> or <component> element) can have a corresponding dispatch <item> element in the appintgcontroller component definition that provides a specific success or failure page. If the menu item does not have an entry in the appintgcontroller component definition, the default success and failure pages are used.

Table 6-1. Application Connectors menu configuration elements

Element	Description
<menugroup>	Content authoring application menu item group. The id attribute specifies the menu configuration ID. Contains one <contentsourcemenueitem>, one or more <actionmenueitem>, <nlsbundle>, and optional <menu> and <menuseparator/> elements.
<contentsourcemenueitem>	Contains one or more <value> elements
<contentsourcemenueitem>.<value>	Contains a string or <nlsid> element describing each source
<menuseparator/>	Displays a separator between menu items
<menu>	Optional. Set of <actionmenueitem> elements that form a submenu for the <menu> item. To remove an entire submenu, remove the <menu> element.
<actionmenueitem>	Content authoring application menu item that contains <aidynamic>, <name>, <value>, and either <action> or <component> plus optional elements that are described below. To remove an item from the menu, remove the <actionmenueitem> element.
.<nlsid>	Optional key to a localized label

Element	Description
.<aidynamic>	Specifies the way in which the state of the authoring application affects the menu item: no_dynamic: item always available aiconnection: item enabled if connected to repository any_content: item enabled if repository or external document is open repository_content: item enabled if repository content is open
.<name>	Specifies the menu item name
.<value>	Contains a string value or <nlsid> element that resolves a string label for the menu item
.<action>	Maps a menu item to a WDK action. Either <action> or <component> must be present in the <actionmenuitem> element. To create a custom application connector action, see Customizing application connector components and actions, page 265 .
.<component>	Maps a menu item to a WDK component. Either <action> or <component> must be present in the <actionmenuitem> element. To create a custom application connector component, see Customizing application connector components and actions, page 265 .
.<arguments><argument>	The values of the name and value attribute on the <argument> element are passed to the WDK action. The name must be a defined parameter for the action, and the value must be a valid value for that parameter.

Adding custom menu items to all applications

To add a custom menu item:

1. On the WDK application server machine, copy the `/webcomponent/config/library/appintgmenubar/appintgmenubar_menugroup.xml` file to your application `custom/config` directory.
2. Extend the WDK version of this component by changing the `<menugroup>` element as follows:

```
<menugroup id="appintgmenubar" extends="
  appintgmenubar:webcomponent/config/library/appintgmenubar/apptintgmenubarmenugroup
```

3. Add an `<actionmenuitem>` element in the desired location in the `appintgmenubar_menugroup.xml` file.
See [Table 6–1, page 262](#) for a description of the child elements that comprise the `<actionmenuitem>` element.
4. Save your custom `appintgmenubar_menugroup.xml` file and restart the application server.

Restricting menu items to specific applications

You can restrict menu items to specific applications—that is, remove them from or add them to specific applications.

For example, if your users use lifecycles for Word documents but not for Excel and PowerPoint documents, then you can remove the Lifecycle menu items from Microsoft Excel and PowerPoint, but keep it in Microsoft Word.

To restrict a menu item to specific applications:

1. On the WDK application server machine, copy the file `/webcomponent/config/library/appintgmenubar/appintgmenubar_menugroup.xml` to the WDK application `custom/config` directory.
2. Extend the WDK version of this component by changing the `<menugroup>` element as follows:

```
<menugroup id="appintgmenubar" extends="
  appintgmenubar:webcomponent/config/library/appintgmenubar/apptintgmenubarmenugroup
```

3. Open `app.xml` in `/wdk` and locate the element `<environment>.<clientenv_structure>.<branch>.<parent>` with the value `appintg`. In the `<children>.<child>` elements, find your application string that matches the Microsoft application, for example, `msword` for Microsoft Word.

- To remove a menu item for specific applications only, edit your extended `appintgmenubar_menugroup.xml` file. Enclose the `<actionmenuitem>` element in a `<filter>` element using the `not` keyword, specifying application name strings, separated by commas. (You can identify the menu item by finding the string in `<menuitem>.<value>` and matching it to the menu item name displayed on the menu.)

For example, to remove the **Create New from Template** menu item for Excel and Word applications:

```
<filter clientenv="not msword, not msexcel">
  <actionmenuitem>
    <aidynamic>no_dynamic</aidynamic>
    <name>new_from_template</name>
    <value>
      <nlsid>MSG_NEW_FROM_TEMPLATE</nlsid>
    </value>
    <action>appintgnewfromtemplate</action>
    <arguments>
      <argument name="contentType" value="contextvalue"/>
    </arguments>
  </actionmenuitem>
</filter>
```

- To limit a menu item to specific applications, in the copy of `appintgmenubar_menugroup.xml`, enclose the `<actionmenuitem>` element in a `<filter>` element that specifies the desired application name strings, separated by commas.

For example, to add the **Create New from Template** menu item for Word and Excel:

```
<filter clientenv="msword, msexcel">
  <actionmenuitem>
    <aidynamic>no_dynamic</aidynamic>
    <name>new_from_template</name>
    <value>
      <nlsid>MSG_NEW_FROM_TEMPLATE</nlsid>
    </value>
    <action>appintgnewfromtemplate</action>
    <arguments>
      <argument name="contentType" value="contextvalue"/>
    </arguments>
  </actionmenuitem>
</filter>
```

- Save your custom `appintgmenubar_menugroup.xml` file and restart the application server.

Customizing application connector components and actions

These topics are included:

- [Overview, page 266](#)

- [List of application connector components and actions, page 266](#)
- [Adding application connector components and actions, page 266](#)
- [appintgcontroller component, page 267](#)
- [Managing events, page 270](#)
- [Managing authentication, page 271](#)

Overview

All Documentum menu selections in the content authoring application are dispatched through the Application Connectors controller component `appintgcontroller`. This component launches an action or component via a URL and adds messages and return listeners so that it can return messages, errors or results to the native modal dialog window. If authentication is required, the `appintgcontroller` component jumps to the `appintgcontrollerlogin` component.

All application connector components and actions are described in *Web Development Kit Reference Guide*.

List of application connector components and actions

All Application Connectors component and action names and configuration files are prepended with the text, `appintg`.

Application Connectors components are:

`appintgcontroller`, `appintgcontrollerlogin`, `appintghelp`, `appintgnewfromtemplate`, `appintgopen`, `appintgopenfrom`, `appintgopenfromcabinetslocator`, `appintgopenfromcategorieslocator`, `appintgopenfromlocatorcontainer`, `appintgopenfrommyfileslocator`, `appintgopenfromrecentfileslocator`, `appintgopenfromsubscriptionslocator`, `appintgsaveascabinets`, `appintgsaveascategories`, `appintgsaveascontainer`, `appintgsaveasmyfiles`, `appintgsaveasrecentfiles`, `appintgsaveassubscriptions`,

Application Connectors actions are:

`appintgnewfromtemplate`, `appintgopenfrom`, `appintgsaveas`,

Adding application connector components and actions

In general, you add application connector components and actions in the same way as other WDK components and actions. You can create a completely new component or

action or extend an existing one. Extending an existing component or action means that the new component or action inherits the existing component or action's configuration file <component> or <action> element's child elements.

To create a new Application Connector component or action:

1. Create or extend a component or action definition.
2. Remove or add configuration structures or JSP user interface elements based on your application connector using the <filter> element.
3. To display the new component's own success and failure JSP pages, extend the appintgcontroller component definition and add the new component's success and failure JSP pages in the appintgcontroller component configuration file.

For example, for the appintgnewmenu menu item, the shownewcomponentpage success page is specified, but no failure page is specified, so the default failure page in <pages><failure> is used.

```
<item>
  <name>appintgnewmenu</name>
  <type>action</type>
  <successpage>shownewcomponentpage</successpage>
</item>
```

You could add your own failure page in your extended component definition:

```
<item>
  <name>appintgnewmenu</name>
  <type>action</type>
  <successpage>shownewcomponentpage</successpage>
  <failurepage>myfailurepage</failurepage>
</item>
```

4. Either add a new menu item or modify an existing one that causes the appintgcontroller to execute and display your component.
Refer to [Restricting menu items to specific applications, page 264](#) and [Adding custom menu items to all applications, page 264](#) for more information.
5. Optionally, handle or fire application connector browser or WDK action completion events.
Refer to [Managing events, page 270](#) for more information.

appintgcontroller component

The appintgcontroller component consists of the AppIntgController servlet and JavaScript. This component performs these functions:

- Sets the locale, theme, Webtop view, and current repository if they are defined in the component arguments or stored in a cookie. If they are not in the argument list or a

cookie, the default values as defined in the `appintgcontroller` component definition are used.

- Enables login: Checks for connection requirement and, if required, sets a menu version event and forces login. Provides the `loginas` action that disconnects from all repositories and allows the user to log in with different credentials.
- Dispatches a JSP page or action as specified in the `<type>` element:

Value of `page` in `<type>` element dispatches an event and, if the dispatch item has a `<page>` element, loads a named JSP page from the component definition in a native modal dialog window.

Value of `action` in `<type>` element dispatches the action, adds a return listener, and displays the success or failure page after action completion. Adds context arguments to a menu action map that is downloaded to the client.

If the menu item does not have a `<dispatchitems>` entry, the action or component is dispatched and the default success or failure page is used.

The `appintgcontroller` dispatches menu items from the menu that is named in the `<menugroupid>` element of the `appintgcontroller` component definition. Each action or component that is defined as a menu item in the `menugroup` definition (in `appintgmenubar_menugroup.xml`) can have a corresponding entry of type "action" in the `appintgcontroller` component definition. This allows you to configure a specific success or failure page for the action. For example, for the `appintgnewfromtemplate` menu item, there is an item in the controller definition (in `apptgcontroller_component.xml`) that specifies a success page:

```
<dispatchitems>
  <item>
    <name>appintgnewfromtemplate</name>
    <type>action</type>
    <successpage>opendocumentevent</successpage>
  </item>
</dispatchitems>
```

Table 6–2, page 268 describes the configuration elements for dispatching actions or components.

Table 6-2. Appintgcontroller <dispatchitems> elements

<code><item></code>	Contains one of each: <code><name></code> , <code><type></code> , and either <code><page></code> or <code><successpage></code>
<code>.<name></code>	Contains the name of a menu action, if the type is action. Contains the name of an event, if the type is page.
<code>.<type></code>	Specifies the type of menu item to dispatch: action or page

.<page>	Specifies a named child element of <pages>, to launch a page
.<successpage>	Optional. Specifies a named child element of <pages>, to launch a success page after action completes. If this element is not present, the default success page in <pages><success> is used.
.<failurepage>	Optional. Specifies a named child element of <pages>, to launch a failure page after action fails. If this element is not present, the default success page in <pages><success> is used.

When the user selects a menu item, the `appintgcontroller` builds a URL, and the requested page loads in a modal dialog window that hosts a browser control within your application. If the menu item maps to an action, the WDK action dispatcher performs the action and returns the success or failure JSP page to the modal dialog window. If the menu item maps to a component, the WDK component dispatcher returns the component JSP start page to the modal dialog window.

Each JSP page that is named for a dispatch item must be defined in the `appintgcontroller` component definition. For example, the `appintgopendocumentevent` item specifies a value of `opendocumentevent` for the `<successpage>` element. The element `<pages>.<opendocumentevent>` defines this page and has a value of `/webcomponent/library/appintgcontroller/appIntgOpenDocumentEvent.jsp`, which specifies the path to the page to be loaded.

The following default pages are defined within the `<pages>` element of the component definition. If an item does not specify a success or failure page, then the default page is used:

Table 6-3. Required pages in `appintgcontroller` component definition

<start>	Specifies the path to an error handler page that is loaded if the action or component is not dispatched
<disconnect>	Specifies the path to a page that is used to disconnect existing sessions and return to the controller for new login. This page invalidates the HTTP session.

<success>	Specifies the path to the default success page that is loaded after the action completes successfully
<failure>	Specifies the path to the default failure page that is loaded after the action fails to complete

Managing events

The browser control registers an event listener to respond to WDK component client events.

WDK components generate events for AppConnectors. The firing of events are configured in the JSP page using control tags. Three types of events can be fired:

- Cross-integration events, which communicate state between applications
- Browser events, which communicate client-side JavaScript execution
- WDK action completion events, which communicate server-side state changes

The following controls fire AppConnectors browser events:

- <dmf:body showdialogevent="true" ...>
Fires the showdialogevent when a JSP page is rendered, to open a modal dialog window in the client application and display the component or container start page. This event is not needed in the component JSP page if the container JSP page fires the event. For example, refer to /wdk/system/changepassword/changepassword.jsp.
- <dmf:fireclientevent>
Fires a client event on page rendering that is handled on the client. For events that are handled by AppConnectors, an aiEvent is fired and passed to the AppConnectors. The actual event name is passed in a <dmf:argument> element that is contained within <dmf:fireclientevent>. The AppConnectors handle the event in the Windows client application.
- <dmf:firepresubmitclientevent>
Fires an aiEvent before submitting the control onclick event to the server.

Application event handlers for showdialogevent and aiEvent are defined in a .js file. The file is included in the JSP page as follows:

```
<script language="JavaScript" src="
  <%=strContextPath%>/wdk/include/appintgevents.js">
</script>
```

The following action completion events can be fired by WDK components:

- Success: LoginSuccess, CheckinSuccess, CheckoutSuccess

- ShowMessage
- OpenDocument: arguments filenamewithpath, object Id, objectType, contentType, lockOwner, folderId, actionMap

Managing authentication

For a content authoring application that has a connector installed, the authentication scope is read at startup from the .config file, for example, winword.exe.config. The following manual authentication scopes are supported:

- system
Allows a single user to connect to any repository in more than one content authoring application. Credentials are stored as a singleton.
- process
Allows a single user to connect to any repository within the same application process. Credentials are stored separately for each content authoring application.
- none
Allows each user to connect to a repository within the process. Different processes can create repository sessions for different users. Credentials are not stored, and timeout of the HTTP session causes a dialog box to be shown. Other authentication schemes such as single sign-on do not use the Credential Service and have a scope of "none."

For the first two authentication scopes, the user credentials are stored securely in the Credential Service, a Windows service executable that runs under Service Control Manager.

The manual authentication scope is configured in the .config file, in the <manualAuthenticationScope> element, similar to the following:

```
<wdkAppLocalInfo name="webtop" ...>
  <host>http://myserver:port/webtop</host>
  <folder>webtop2</folder>
  <manualAuthenticationScope>system</manualAuthenticationScope>
</wdkAppLocalInfo>
```

The AppConnectors login page, appintglogin.jsp, fires the client event aiEvent when the page loads, which brings up a native modal dialog window that hosts a browser control for login to the application.

The default height and width of the login window can be configured in the <dmf:fireclientevent> tag on the login JSP page. Users can increase the size by dragging it, and the new size will be used the next time the same component is launched. The default size that is specified in the JSP page becomes the minimum size for the modal dialog window.

Changing the application server for the AppConnectors client

The 5.3 release supports a single content source, that is, a WDK-based application as the gateway for content from a Documentum repository. You must manually edit the app.config files of an Application Connector to change the active content source.

1. Locate the app.config files, for example, WINWORD.exe.config, EXCEL.exe.config, POWERPNT.exe.config. These files are located in two places. The default installation location for Microsoft Office files is %ProgramFiles%\Microsoft Office\OFFICE{10,11} and %ProgramFiles%\Microsoft Office\OFFICE{10,11}\Documentum. Both files in a pair must be modified together.
2. Locate the <wdkAppLocalInfo element>.
3. Change the <host> subelement value to the URL to the desired instance, for example, <http://localhost:8080/webtop>

The user can choose a different instance of an application server content source and save it as a user preference.

Configuring Preferences

Component preferences can be used to set preferences for all users. These preferences apply to a single component and are exposed in the component definition in order to make them configurable.

Preferences can be exposed to individual users. The user environment preferences are configured in the definitions located in the `/webcomponent/config/environment/preferences` subdirectories.

The following topics describe preference configuration:

- [Preference definition, page 273](#)
- [Configuring default component and user preferences, page 276](#)
- [User column display preferences, page 277](#)
- [Sample preference definitions, page 281](#)

Custom preferences for components or users must be implemented so that the component uses the component or user preference in rendering. Refer to [Chapter 21, Implementing Component and User Preferences](#) for details.

Preference definition

A component can define its preferences within a `<preferences>` element or as a custom element. If the component preference will not be exposed as a user preference, it is simpler to define the preference within a custom element. For example, the `myobjects_drilldown` component defines a preference of displaying the user's folders through a custom element, `<showfolders>`. The value of this element is used to set the preference for all users.

The following table shows elements that can be used within a `<preferences>` element to define preferences for a component:

Table 7-1. <preference> elements

Element	Description
<preference id='name' disabled='true'>	Defines the preference and its required ID attribute. Disabled attribute is optional and defaults to false.
<label>	Required. Sets the display name of the preference.
<description>	If supplied, the description is displayed underneath the main row of subcontrols.
<nlsid>	Inside <label> and <description . If present, the value is resolved from the <nlsbundle> referenced in the component definition file.
<type>	Required. Value type: int string boolean columnlist
<value>	Default value for the preference
<position>	Position of the value control: right (default) left
<display_hint>	Forces the control to use a type of display: password (for strings or integers) dropdownlist listbox hidden. A <constraints> element containing one or more <element> elements must be present for dropdownlist or listbox.
<listbox_size>	If you are using a listbox , use this element to specify how many elements are visible in the listbox at any one time (default = 5).
<constraints>	Preference constraints. If the display_hint tag is absent, the control will look for a <lowerend> and <upperend> for an int type. For other types this element will be ignored. With the display_hint tag present the control will look for <element> instead of <constraints>.
<lowerend>	Lower limit for int. For other types, and when the display_hint element is present, <lowerend> will be ignored.

Element	Description
<upperend>	Upper limit for int. For other types, and when the display_hint element is present, <upperend> will be ignored.
<element>	Element in a list of int or string type. Used with <display_hint> to build a list of elements for a dropdown list or listbox. The value of <element> both the preference value and its description.
<element value="some_value">	Same as <element>, to be used to present a different description from the preference value. The value is in the element attribute, and the description is in the element content..
<editcomponent>	Configures column lists only (<type> columnlist</type>). Specifies the component to launch for editing the preference.
<editcontainer>	Configures column lists only (<type> columnlist</type>). Specifies the container to launch for editing the preference.
<inherits>	Configures column lists only (<type> columnlist</type>). Specifies the preference definition that sets the columns to be displayed. A checkbox will be displayed. If checked, the preference specified in the <inherits> element will be used.

String and integer preferences are displayed in a text box (default), listbox, or dropdownlist. These can be displayed with a set of values to choose from (preference assistance). Boolean preferences are displayed in a checkbox. Column lists or attributes are displayed by the columnlist or attributelist control, respectively.

A hidden preference has the display_type set to hidden and renders a hidden field, without label or description. The hidden preference can be programmatically accessed.

Configuring default component and user preferences

All component preferences that are exposed within a <preferences> element in a component can be configured with default values and selections. Copy the component definition into the /custom/config directory and change the definition to extend the original definition. Configure the values within a <preference> element based on the rules presented in [Preference definition, page 273](#).

For example, to change the number of days for which to display the user's objects in the Webtop streamline view, copy the file myfiles_streamline_component.xml in /webtop/config to /custom/config and change the component element to the following:

```
component id="myfiles_streamline" extends="
  myfiles_streamline:/webtop/config//myfiles_streamline_component.xml"
```

Because the Webtop component doesn't change the preferences of the WDK component definition, you will need to open the WDK definition in myobjects_drilldown_component.xml, located in /webcomponent/config/library/myobjects, and copy the <preferences> element into your new component definition. You can change the value of <modifiedwithindays> similar to the following:

```
<preferences>
  ...
  <preference id="modifiedwithindays">
    <label><nlsid>PREF_LBL_MODIFIED</nlsid></label>
    <description><nlsid>PREF_DESC_MODIFIED</nlsid></description>
    <type>int</type>
    <value>30</value>
  </preference>
</preferences>
```

User preferences are exposed within the definitions in /webcomponent/config/env/preferences. The following table describes the types of preferences that are set by each preferences component:

Table 7-2. User preference components

Webtop Label	Component ID	Description
General	general_preferences	View, entry component, theme, accessibility, checkout location, drag and drop

Webtop Label	Component ID	Description
Columns	display_preferences	Attribute columns to be displayed for each configured component. Refer to User column display preferences, page 277 .
Virtual documents	vdm_preferences	Opening options, bindings, copy, checkout
Login	savecredential	Save or remove login credentials
Repositories	visiblerepository_preferences	Select repositories to be visible
Search	searchsources_preferences	Sets default search location
Formats	format_preferences	Sets preferred rendition, viewing format, and editing format for object type
Debug	debug_preferences	Sets debugging options for development environments. (Usually should be removed for deployment.)

User column display preferences

User display preference settings are configured in the `display_preferences` component definition (`/webcomponent/config/environment/preferences/display/display_preferences_component.xml`). This component has the same elements that are used for defining component preferences ([Preference definition, page 273](#)). Each component can have an entry that gives the user the ability to select columns or other preference settings.

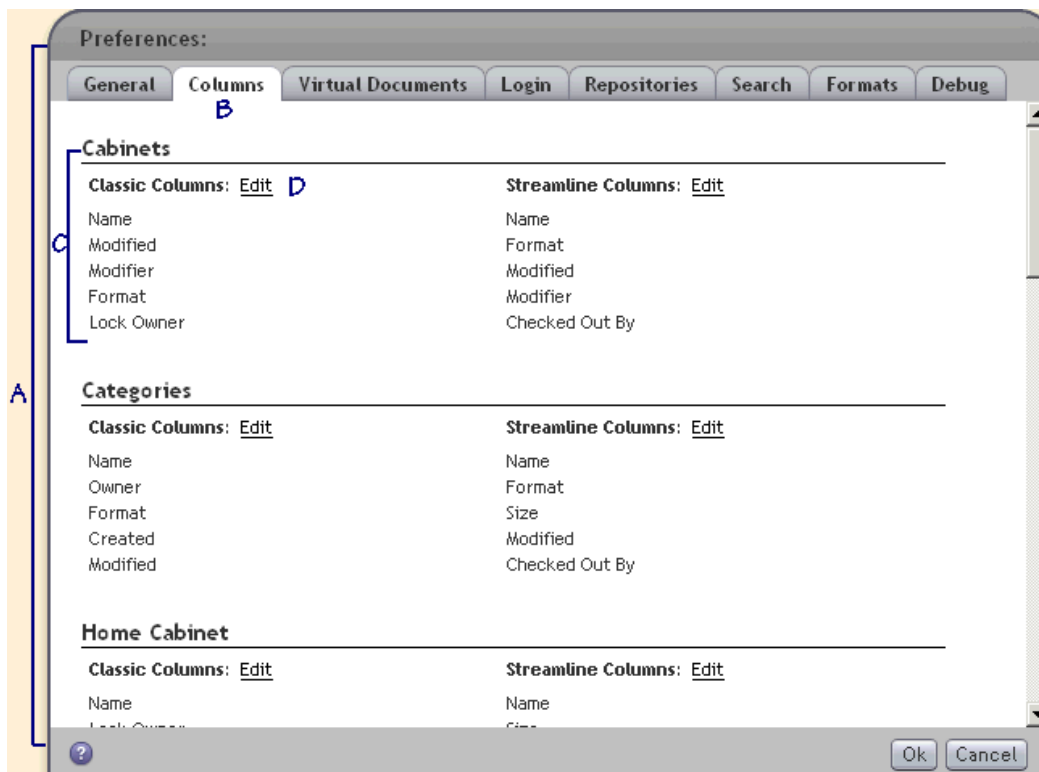
The preferences elements specific to the `display_preferences` component are as follows:

Table 7-3. Column display preference elements

<preference>	ID for the column display preference
<display_docbase_types>	Sets the contents of the dropdown list to specify column preferences for an object type. Contains <docbase_type> elements. You can add this element to any <preference> element in the display_preferences component to override the default object type list.
.<docbase_type>	Adds a repository type to display in the drop-down list of types for setting column display preferences. Contains a <value> element whose value must correspond to a type in the data dictionary and a <label> element that will display a label for the type.
<show_repeating_attributes>	Set to false to not display repeating attributes. Will not affect attributes in the default list or attributes already in the selected list.
<enableordering>	Generates up and down arrows that allow the user to reorder columns. Default = true

The following diagram illustrates the structure of the column display_preferences component in WDK:

Figure 7-1. WDK preferences components

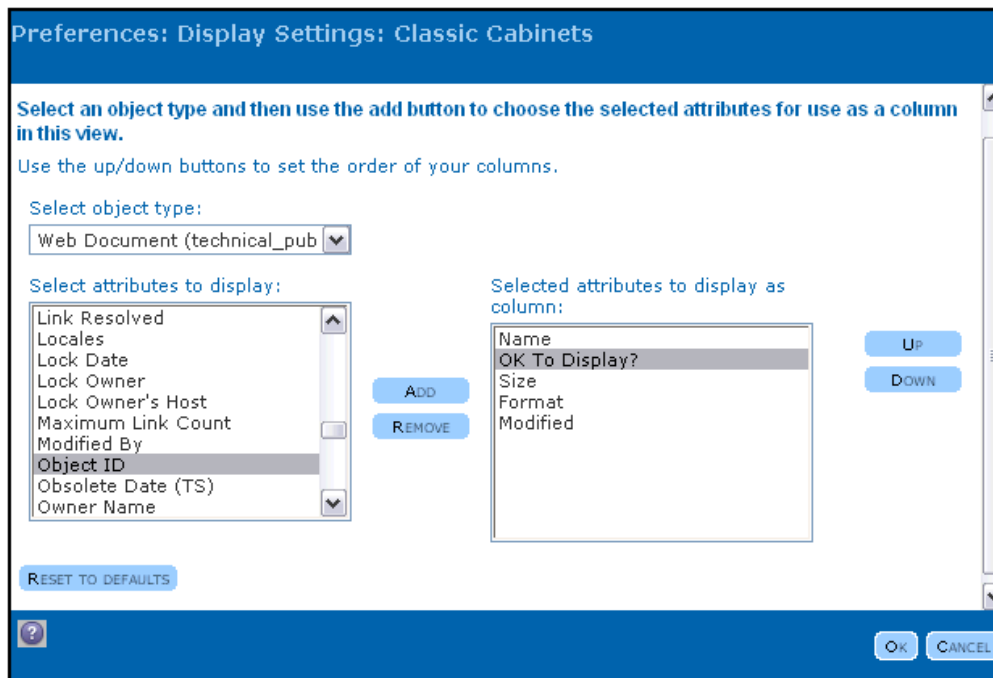


Legend:

- A
Preferences container UI with header title and footer buttons
- B
Hidden preferencescope tag, which sets the scope to 'User'
- C
Generated by <preference> element in this (display_preferences) component definition. The preference element specifies <value> as component[id=doclist].columns, so this line configures the columns that are displayed in the doclist component or a component that extends doclist. The initial list of columns (Name, Size, Format, Modified) is read from the component definition, in this case, the <columns> element of the doclist definition.
- D
The edit link launches the component named in the <editcomponent> element, in the container named in the <editcontainer> element. In the example shown below,

the columnselector component is launched in its container to edit the selected preference. (The attributes are generated by an attributeselector control.)

Figure 7-2. Column selector component



The column selector component UI is generated by <preference> element in this (display_preferences) component definition. The preference element specifies <value> as component[id=drilldown].columns, so this line configures the columns that are displayed in the drilldown component or a component that extends doclist. The initial list of columns (Name, Size, Format, Modified) is read from the component definition, in this case, the <columns> element of the doclist definition.

Example 7-1. Adding a display preference for a custom type

In the following example, a custom type is added so that the user can configure display of custom attributes:

1. Copy the file display_preferences_component.xml from /webcomponent/config to /custom/config.
2. Modify the component definition to add the highlighted extends attribute as follows:

```
<component id="display_preferences" extends="
  display_preferences:webcomponent/config/environment/
  preferences/display/display_preferences_component.xml">
```

3. Add your custom type to the <display_docbase_types> element, similar to the following:


```

<display_docbase_types>
  <docbase_type>
    <value>dm_document</value>
    <label><nlsid>LBL_DOCUMENT</nlsid></label>
  </docbase_type>
  <docbase_type>
    <value>dm_folder</value>
    <label><nlsid>LBL_FOLDER</nlsid></label>
  </docbase_type>
  <docbase_type>
    <value>technical_publications_web</value>
    <label>Web Document</label>
  </docbase_type>
</display_docbase_types>

```

- Refresh the config files in memory by navigating to /wdk/refresh.jsp. The display for column preferences should show the custom type, similar to the following:

Figure 7-3. Custom type column preferences

Preferences: Display Settings: Classic Columns for Cabinets:

Select object type:

Select attributes to display:

Parent Edition
Product Name
Product Version
Public
Publish Formats
Reference
Replica
Resume to state
Retain Content Until
Room

Selected attributes to display as column:

Name
Format
OK To Display?

Sample preference definitions

The following preferences examples show the possible types of preferences and how to define them. Line breaks are added to this example for readability.

```

<config version='1.0'>
  <scope>
    <component id="mycomponent">
      <!-- Other component tags here -->
    </component>
  </scope>
</config>

```

```
<!-- Component-specific preferences -->
<preferences>
  <preference id="test1">
    <label>Test 1</label>
    <description>Test for simple integers</description>
    <type>int</type>
    <value>1</value>
  </preference>
  <preference id="test1a">
    <label>No Description</label>
    <type>int</type>
    <value>9876</value>
  </preference>
  <preference id="test1b">
    <label>No Default Value</label>
    <type>int</type>
    <description>An integer with no default value set</description>
  </preference>
  <preference id="test1c">
    <label>Test 2b</label>
    <description>Integer in a password control.</description>
    <type>int</type>
    <display_hint>password</display_hint>
  </preference>
  <preference id="test2">
    <label>Test 2</label>
    <description>Test for simple strings with user scope set by default
    </description>
    <type>string</type>
    <value>Hello</value>
    <scope>user</scope>
  </preference>
  <preference id="test2a">
    <label>Test 2a</label>
    <description>Test for simple strings with no default value
    </description>
    <type>string</type>
  </preference>
  <preference id="test2b">
    <label>Test 2b</label>
    <description>String in a password control.</description>
    <type>string</type>
    <display_hint>password</display_hint>
  </preference>
  <preference id="test3">
    <label>Test 3</label>
    <description>Test for simple boolean with group scope
    </description>
    <type>boolean</type>
    <value>false</value>
    <scope>group</scope>
  </preference>
  <preference id="prefsString">
    <label>String Options</label>
    <description>Displayed in a dropdown list with "Three"
    selected by default
    </description>
```

```

    <type>string</type>
    <value>Three</value>
    <display_hint>dropdownlist</display_hint>
    <constraints>
      <element>Once</element>
      <element>Twice</element>
      <element>Three</element>
      <element>Four</element>
      <element>Many</element>
    </constraints>
  </preference>
  <preference id="prefsInt">
    <label>Integer Options</label>
    <description>Integer preference with 1, 3, 4, 5, 9 displayed in
      a listbox and 5 selected by default</description>
    <type>int</type>
    <value>5</value>
    <display_hint>listbox</display_hint>
    <constraints>
      <element>1</element>
      <element>3</element>
      <element>4</element>
      <element>5</element>
      <element>9</element>
    </constraints>
  </preference>
  <preference id="prefsStringDesc">
    <label>More String Options</label>
    <description>The elements have separate values and descriptions.
  </description>
    <type>string</type>
    <value>Three</value>
    <display_hint>dropdownlist</display_hint>
    <constraints>
      <element value="Once">Only once</element>
      <element value="Twice">Maybe twice</element>
      <element value="Three">Possibly three</element>
      <element value="Four">A large four</element>
      <element value="Many">Far too many</element>
    </constraints>
  </preference>
  <preference id="prefsIntDesc">
    <label>More Integer Options</label>
    <description>Integer Dropdownlist type with text descriptions
  </description>
    <type>int</type>
    <value>3</value>
    <display_hint>dropdownlist</display_hint>
    <constraints>
      <element value="1">Very Small</element>
      <element value="2">Small</element>
      <element value="3">Medium</element>
      <element value="4">Large</element>
      <element value="5">Extra Large</element>
    </constraints>
  </preference>
  <preference id="prefsConstr1">

```

```
<label>Constrained Integer</label>
<description>Constrained integer preference - not below 3
</description>
<type>int</type>
<value>5</value>
<constraints>
  <lowerend>3</lowerend>
</constraints>
</preference>
<preference id="prefsConstr2">
  <label>Constrained Integer</label>
  <description>Constrained integer preference- notover 29
  </description>
  <type>int</type>
  <value>5</value>
  <constraints>
    <upperend>29</upperend>
  </constraints>
</preference>
<preference id="prefsConstr3">
  <label>Constrained Integer</label>
  <description>Constrained integer preference- not below 3 or above 29
  </description>
  <type>int</type>
  <value>5</value>
  <constraints>
    <lowerend>3</lowerend>
    <upperend>29</upperend>
  </constraints>
</preference>
<preference id="listboxsizetest">
  <label>String Listbox with listbox_size of 10</label>
  <type>string</type>
  <value>A01</value>
  <display_hint>listbox</display_hint>
  <listbox_size>10</listbox_size>
  <constraints>
    <element>A01</element>
    <element>B02</element>
    <element>C03</element>
    <element>D04</element>
    <element>E05</element>
    <element>F06</element>
    <element>G07</element>
    <element>H08</element>
    <element>I09</element>
    <element>J10</element>
    <element>K11</element>
    <element>L12</element>
    <element>M13</element>
    <element>N14</element>
    <element>O15</element>
  </constraints>
</preference>
<preference id="disabled1" disabled="true">
  <label>Disabled Simple Integer</label>
  <type>int</type>
```

```

    <value>9876</value>
  </preference>
  <preference id="disabled2" disabled="true">
    <label>Disabled Simple String</label>
    <type>string</type>
    <value>Disabled</value>
  </preference>
  <preference id="disabled3" disabled="true">
    <label>Disabled Simple Boolean</label>
    <type>boolean</type>
    <value>true</value>
  </preference>
  <preference id="Disabled4" disabled="true">
    <label>Disabled string Listbox</label>
    <type>string</type>
    <value>Three</value>
    <display_hint>listbox</display_hint>
    <constraints>
      <element value="Once">Only once</element>
      <element value="Twice">Maybe twice</element>
      <element value="Three">Possibly three</element>
      <element value="Four">A large four</element>
      <element value="Many">Far too many</element>
    </constraints>
  </preference>
  <preference id="Disabled5" disabled="true">
    <label>Disabled Integer Dropdownlist</label>
    <type>int</type>
    <value>3</value>
    <display_hint>dropdownlist</display_hint>
    <constraints>
      <element value="1">Very Small</element>
      <element value="2">Small</element>
      <element value="3">Medium</element>
      <element value="4">Large</element>
      <element value="5">Extra Large</element>
    </constraints>
  </preference>
</preferences>
  <!-- Other component tags here -->
</component>
</scope>
</config>

```


Configuring Roles and Client Capability

Roles are defined in Content Server 5.1 and above as a special type of group. The `group_class` attribute on the group is set to 'role', and the `group_name` attribute is set to the role name. (Refer to the Server administration guide for more information on role groups.)

Webtop and WDK components can be configured to use any role that is defined in the repository. If no roles for the user are configured in the repository, or if the repository is pre-5.1, Webtop defaults to using the client capability model with the following `client_capability` attributes on the `dm_user` object: consumer, contributor, coordinator, and administrator (refer to [Client capability plugin, page 289](#)). If the user has no client capability level set, the role service assigns the user the consumer role.

Role configuration is described in the following topics:

- [Role configuration overview, page 287](#)
- [Client capability plugin, page 289](#)
- [Docbase role plugin, page 291](#)
- [Role-based actions, page 292](#)
- [Role-based filters, page 293](#)
- [Role-based UI, page 294](#)

The role model APIs and role model customization are described in [Chapter 16, Customizing Roles](#).

Role configuration overview

The client application, such as Webtop, Web Publisher, or your custom application, enforces role capabilities through configuration. You can configure roles in WDK to perform the following role-based presentation:

- Role-based actions

You can restrict actions to users who belong to a specified role. For example, you can restrict the checkout action in the action definition to contributors or higher only, for example:

```
<precondition class="com.documentum.web.formext.action.RolePrecondition">
```

```
<role>contributor</role>
</precondition>
```

Refer to [Role-based actions, page 292](#) for more information.

- Role-based filters

You can restrict a container so that it displays some components to specified roles. For example, you can filter the components in the folder properties container to display the permissions component only to administrators or higher, while all roles have access to the remaining components in the container:

```
<scope type="dm_folder">
  <component id="properties" extends="type='*' application='webcomponent'">
    <contains>
      <filter role="administrator">
        <component>permissions</component>
      </filter>
      <component>discussions</component>
      <component>locations</component>
      ...
    </contains>
  </component>
</scope>
```

For more information, refer to [Role-based filters, page 293](#). For a general description of the use of filters in configuration, refer to *Web Development Kit and Client Applications Development Guide*.

- Role-based component UI

You can present a different UI to different roles. For example, a different properties page can be displayed to administrators and general users:

```
<scope type="dm_folder", role="administrator">
  <component id="properties" extends="type='*' application='webcomponent'">
    <contains>
      <component>permissions</component>
      <component>discussions</component>
      <component>locations</component>
      ...
    </contains>
  </component>
</scope>
```

Refer to [Role-based UI, page 294](#) for more information.

The role service gets the user's role using either the repository role model plugin (refer to [Docbase role plugin, page 291](#)), the client capability plugin (refer to [Client capability plugin, page 289](#)), or both (refer to [Docbase role plugin, page 291](#)). A role is then used in one the following ways, depending on how it was called:

Role is defined as a scope on an action	The action service limits the action to users having the specified role or a parent of the role.
Role is defined as a filter in a container	The configuration service displays the filtered component to users who have the specified role or a parent of the role
Role is defined as a scope on a component	The configuration service displays the component that is defined for the role or a parent of the role

Client capability plugin

The client capability role model has four fixed roles: consumer, contributor, coordinator, and administrator. The user's role is determined by the value of the `client_capability` attribute on the user's `dm_user` object. The following values of this attribute map to the client capability roles:

consumer	1
contributor	2
coordinator	4
administrator	8

If roles are not specified as special groups in the repository, or if the repository is pre-5.1, or if the user is not assigned to any role group, the client capability model is used to define the user's fallback role. If the user does not have a value assigned to the `client_capability` attribute, the model will default the user to a consumer role.

The operations that can be performed by each client capability role in a default Webtop configuration are described in the table below. Capabilities are cumulative: if an operation is available to a user role, it is available to higher roles as well:

Table 8-1. Client capability roles

Operation	Consumer	Contributor	Coordinator	Administrator
Create folder		X	X	X
Create cabinet			X	X

Operation	Consumer	Contributor	Coordinator	Administrator
Create object		X	X	X
Import		X	X	X
View object	*	*	*	*
Check in		X	X	X
Check out		X	X	X
Edit		X	X	X
Change virtual doc		X	X	X
Delete		X	X	X
Change properties	*	*	*	*
Rename object		X	X	X
Search repository	*	*	*	*
Send to distribution list	*	*	*	*
Lifecycle operations	*	*	*	*
Router/workflow participant	*	*	*	*
Add to clipboard		X	X	X
Webtop Admin node				X

* Operations that are not scoped by client capability role. Some of these actions check permissions in the launch or precondition class, or the component checks repository permissions, but the action or component does not specifically check the client capability level.

Docbase role plugin

Content Server version 5.2.5 and above stores a list of roles that are valid for each user. A role is a `dm_group` object whose `group_class` attribute is set to `role` and whose `group_name` attribute defines the name of the role. Role groups can be members of other role groups, and a user can be member of more than one group. (For information on how to set up roles, refer to *Content Server Administration Guide*.)

Role groups can be grouped together into a domain group, which is a `dm_group` object whose `group_class` attribute is set to `domain`. A domain provides a way to limit the groups of roles that are seen by the application. Along with client capability mapping in `app.xml`, you can create a simple hierarchy of roles that ignore the user's roles in another application. For example, a user can have an administrator role in one domain and a contributor role in another domain.

You can specify the domains that are supported by your application in the `<rolemodel>.<domain>` setting in `app.xml`. If no domains are specified in `app.xml`, the default gets roles in any domain in the repository.

The Docbase role model is the default role plugin for WDK client applications. When a role is used in an action or component definition, the role qualifier is handled by the role service, which calls the Docbase role plugin. This plugin queries the repository for a list of the current user's roles and for the super roles of the user's roles. The query is reissued every ten minutes. This allows the application to dynamically update a cache of repository roles and their hierarchy and the user's assigned roles.

By default, if roles are not defined in the repository or the user has not been assigned a role, the Docbase role model plugin falls back to the client capability plugin. This behavior can be disabled by setting `<rolemodel>.<client_capability_fallback_enabled>` to `false` in `app.xml`.

Custom roles are mapped to client capability in your custom `app.xml`, forming a role hierarchy for the application. (Refer to `<rolemodel>` element, page 64 for more information on the configuration elements.) You can use both custom roles and client capability roles in your application. For example, your repository has the following role hierarchy created by adding the `sitemanager` role to the `pagedeveloper` role using a tool such as Documentum Administrator:

```
sitemanager
  pagedeveloper
```

In `/custom/app.xml`, you map these roles to client capability as follows:

```
<rolemodel>
  ...
  <client_capability_aliases>
    <administrator_roles>
      <role>sitemanager</role>
    </administrator_roles>
```

```

    <coordinator_roles>
      <role></role>
    </coordinate_roles>

    <contributor_roles>
      <role>pagedeveloper</role>
    </contributor_roles>

    <consumer_roles>
      <role></role>
    </consumer_roles>
  </client_capability_aliases>
  ...
</rolemodel>

```

A user whose role is pagedeveloper will have the following roles:

```

  pagedeveloper
  contributor
  consumer

```

A user whose role is sitemanager will have the following roles:

```

  sitemanager
  administrator
  coordinator
  pagedeveloper
  contributor
  consumer

```

If you disable client capability fallback in app.xml, the user will have only the roles that are set up in the repository. You must then make sure that no client capability roles are used in your action and component definitions. If you disable client capability fallback but map user roles to client capability roles in app.xml, client capability will still be used.

Role-based actions

There are two ways you can base actions on roles:

- Actions can be performed only by certain roles

For example, you may wish to restrict checkout of documents even though the user has VERSION permission. This is applied through a role precondition. In the following example, only contributors can check out documents:

```

<action id="checkout">
  <params>
    <param name="objectId" required="true"></param>
    <param name="lockOwner" required="false"></param>
    <param name="ownerName" required="false"></param>
  </params>
  <preconditions>

```

```

    <precondition class="com.documentum.web.formext.action.
      RolePrecondition">
      <role>contributor</role>
    </precondition>
    <precondition class="com.documentum...CheckoutAction"/>
  </precondition>
</preconditions>
</action>

```

The RolePrecondition class checks to make sure that the user belongs to the named role before the action can be executed.

Note: Disabled actions can be invisible or shown as disabled.

- Actions cannot be performed by certain roles

You can make actions invisible to users by adding a role scope to the action definition. The UI feature that launches the action, such as a button or menu item, will be invisible if the user does not have a role within the scope of the action. In the following example, the consumer role cannot perform the edit action:

```

<scope role="consumer">
  <action id="editfile" notdefined="true"></action>
  <!-- list here the actions that a consumer can perform -->
</scope>

```

The action service queries the role service to determine whether an action can be executed based on the user's role. The action precondition is evaluated together with ACL permissions on an object. The action service will allow actions to execute, even if the user does not have a required role, in the following cases:

- Operations on an object that is owned by the user
- Operations by a repository superuser, who may need to force access to objects as a part of repository administration

Role-based filters

A filter element restricts the contained element to the specified value of the filter attribute. When the filter attribute is role, the filter is applied to users having the specified role. For example, a container component can restrict users to certain components within the container with a <filter> key. In the following example from a properties container definition, only administrators or higher will have access to the permissions component and can change permissions on folders, while the other contained components are available to all users:

```

<scope type='dm_folder'>
  <component id="properties" extends="
    propertysheetcontainer:wdk/config/propertysheetcontainer_component.xml">
    ...
  <contains>
    <filter role="administrator">

```

```
    <component>permissions</component>
  </filter>
  <component>attributes</component>
  <component>history</component>
  </contains>
  ...
</scope>
```

Role-based UI

You can make entire components available based on roles. In this case, you would use the role attribute of the component scope element. The following example makes the properties of a folder accessible only to site managers. Users with other roles will see the default properties component for `dm_sysobject`:

```
<scope type='dm_folder' role='sitemanager'>
  <component id="properties" extends="type='*' application='webcomponent'">
    <contains>
      <component>permissions</component>
      <component>discussions</component>
      <component>locations</component>
      <component>annotations</component> </contains>
    </component>
  </scope>
```

Configuration Examples

The following examples describe some of the most commonly configured components and features in a WDK-based Web application:

- [Configuring buttons and images, page 295](#)
- [Configuring dynamic menu items, page 300](#)
- [Configuring content display, page 302](#)
- [Configuring navigation base cabinet or folder, page 305](#)
- [Configuring branding, page 307](#)
- [Configuring validators, page 308](#)
- [Configuring application startup, page 309](#)
- [Configuring the properties container, page 311](#)
- [Configuring attributes, page 312](#)
- [Creating a custom object filter, page 317](#)

Configuring buttons and images

This topic describes how to add and configure buttons and images:

- [Adding a button or image, page 296](#)
- [Changing a button label, page 296](#)
- [Changing a button or image style, page 297](#)
- [Changing button or image function, page 298](#)

The following examples will show some of the types of configuration listed above. For other examples, refer to the `/wdk/samples/control_pen/buttonPen.jsp` page.

Adding a button or image

To add a button or image, use the JSP button or image tag from the dmf tag library in the appropriate location in the HTML block of your JSP. The following example adds a **Close** button within an HTML table cell:

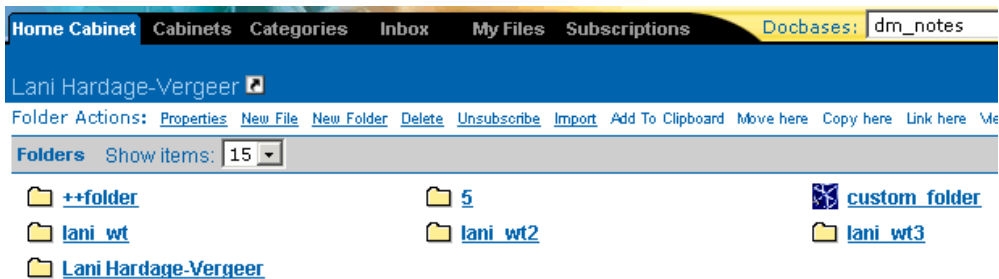
```
<td>
  <dmf:button cssclass='buttonLink' nlsid='MSG_CLOSE'
    onclick='onClose' default='true' height='16'
    imagefolder='images/dialogbutton' />
</td>
```

Example 9-1. Adding an icon for a custom type

You can provide icons for your custom object, cabinet, or folder types. Create a 16x16 pixel icon for each custom type. Create a type folder for each theme that is supported by your application, and place a theme-appropriate icon in the folder. The folder must have the following path: `/custom/theme_name/icons/type` where `theme_name` is the theme for which you wish the icon to be used.

The icon file must be named with the custom object type name and with a `t_` prefix and a `_16` postfix. For example, if your custom type is named `my_sop`, the icon would be named `t_my_sop_16.gif`. An example of a custom folder icon is shown below. Folders of type `dm_folder` have the default folder icon, and the folder of type `km_task` has a custom icon. The custom icon file name is `t_km_task_16.gif`:

Figure 9-1. Custom type icon



Note: The `DocbaseIconTag` class checks for a format icon first. If it does not find one for the object's format, it then checks for a type-specific icon.

Changing a button label

You can change button content with the following attributes:

- label
- nlsid

datafield
 imagefolder
 tooltip

Button labels are specified in the `nlsid` attribute, which overrides a label attribute value. To change a label for a specific control in a JSP page, you can add a label attribute and value and remove the `nlsid` attribute. If your button text will be localized, you must use the `nlsid` attribute and add a localized properties file for the button.

The example in [Adding a button or image, page 296](#) displays a text label on the button. The following example overrides the `nlsid` label string by removing the `nlsid` attribute and adding a label attribute:

```
<dmf:button cssclass='buttonLink' label='Finished'.../>
```

To use a different image for the button, specify the path to the image in the `imagefolder` attribute values. The `imagefolder` points to the directory that contains the specific button images, which must be named `left.gif`, `right.gif`, and `bg.gif`. The branding system appends the theme name to the base directory path. The following example points to new images in our custom theme directory:

```
<dmf:button ...imagefolder='images/finishbutton' />
```

Changing a button or image style

You can change button style with the following attributes:

- style and CSS class
- disabled style and CSS class
- width and height
- align

Globally changing the style of a button or image — To change the style of a button or image everywhere it appears in your application, apply your changes in a style sheet. For example, the `webcomponents.css` file in the `coolblue` theme defines a `drilldownHeader` class that references a background image:

```
.drilldownHeader { BACKGROUND-COLOR: transparent;
  BACKGROUND-IMAGE: url('../images/streamline/tabbarbg.gif') }
```

To change the image that is used every time the CSS class is applied to a control, extend the `coolblue` theme, copy the `webcomponent.css` file to your `/theme/css` directory, and change the URL to the background image.

Changing the style of a button or image in a JSP page — The example below overrides the button disabled style in a specific location in the application. The button's disabled style is initially specified as `buttonDisabledLink`:

```
<dmfx:button name='checkout' action='checkout' label='Files'  
  cssclass="buttonLink" disabledclass="buttonDisabledLink" ...>
```

The style is defined in `webforms.css`:

```
.buttonDisabledLink  
{  
  color: #999999;  
  font-family: Arial,Helvetica,sans-serif;  
  font-size: 11px;  
  font-weight: bold;  
}
```

You can specify a CSS class that is defined in your custom CSS file:

```
<dmfx:button name='checkout' action='checkout' label='Files'  
  cssclass="buttonLink" disabledclass="buttonGoneLink" ...>
```

You have defined the custom class in `myCSSstyles.css`:

```
.buttonGoneLink  
{  
  color: #666666;  
  font-family: Comic,Helvetica,sans-serif;  
  font-size: 11px;  
  font-weight: normal;  
}
```

You can override specific properties of the CSS class in the `style` attribute. In the following example, the disabled class is retained and only the font-weight is overridden:

```
<dmfx:button name='checkout' action='checkout' label='Files'  
  cssclass="buttonLink" disabledclass="buttonDisabledLink"  
  style="font-weight:normal" ...>
```

Changing button or image function

You can change button behavior with the following attributes:

- onclick
- visible
- enabled
- runatclient and client event handler
- default

The `onclick` attribute specifies the event handler that will handle the button or image `onclick` event. You can handle the `onclick` event in the browser by setting the `runatclient` attribute to `true`, setting the `onclick` event handler, and then adding an event handler

of that name in the JSP. In the following example, the titlebar page has a company logo image that the user clicks to view an about page:

```
<dmf:image onclick='onClickLogoEvent' runatclient='true'
  src="images/titlebar/logo_16.gif"/>
```

The event handler calls a nested component URL, which displays an information page:

```
<script>
  function onClickLogoEvent(keys)
  {
    postComponentNestEvent(null, "about", "content");
  }
</script>
```

You can change the event handler to navigate to the company Web site:

```
<script>
  function onClickLogoEvent(keys)
  {
    window.top.location.replace("http://www.mycompany.com");
  }
</script>
```

Configuring dynamic buttons and images

This topic describes the configuration of dynamic buttons. You can configure the same attributes as static buttons and images, with some additional configuration of dynamic attributes. These attributes are described in *Web Development Kit Reference Guide*:

- showifdisabled
- showifinvalid
- dynamic
- action

You can change a dynamic button or image to work only if a single item on the page is selected by setting the dynamic attribute to a value of `singleselect`. You can specify that the button or image will work only if no items are selected by setting the value of dynamic to `genericnoselect`. You can set the button or image action to work regardless of the number of items selected on the page using the dynamic value `'generic'`.

In the following example, the menu bar contains a number of menu items that can only be selected when no object in the page is selected:

```
<dmfx:actionmenuitem dynamic='genericnoselect' name='file_newdocument'
  nlsid='MSG_NEW_DOCUMENT' action='newdocument' showifinvalid='true'/>
```

You can change the menu behavior to show the menu item regardless of the items checked by specifying a value of `generic` for the dynamic attribute:

```
<dmfx:actionmenuitem dynamic='generic' name='file_newdocument'
  nlsid='MSG_NEW_DOCUMENT' action='newdocument' showifinvalid='true'/>
```

Configuring dynamic menu items

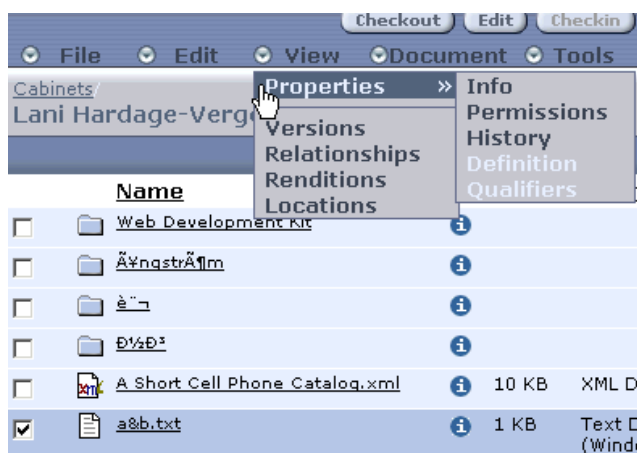
This section describes how to remove a dynamic menu item.

Menu items are dynamically generated, based on object type. The menu itself is configured in a JSP and generated by the menu tag class.

Example 9-2. Removing a menu option

The following example removes a menu option for documents. The quickest way to track down the source of a menu is to right-click in the area of the screen that the menu appears and select View source. In this example, the View menu is contained within menubar.jsp:

Figure 9-2. Webtop view menu



In the Webtop JSP page that contains the menu (menubar.jsp), the View menu is specified by a tag library tag `<dmf:menu name='view_menu'...>`. In order to customize this menubar component, create a copy of the menu JSP page in the /custom directory, for example, /custom/components/menubar.jsp. Create a copy of the component configuration file in the custom configuration directory, for example, /custom/config/menubar_component.xml. The remainder of this example customizes these two files.

In the custom menubar component configuration file (/custom/config/menubar_component.xml), change the component tag to:

```
<component id="menubar" extends="menubar:/webtop/config/menubar_component.xml">
```

Change the `<start>` page definition to:

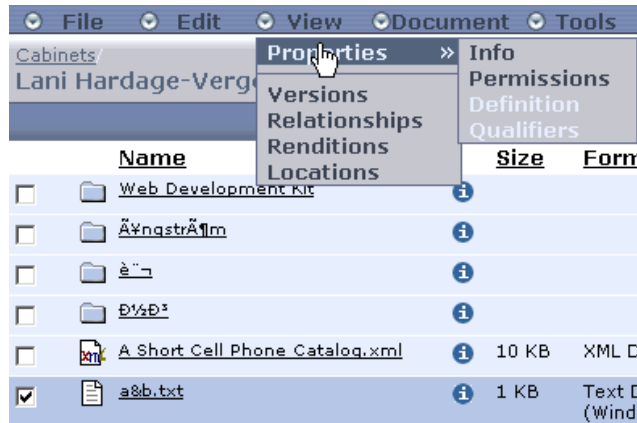
```
<start>/custom/components/menubar.jsp</start>
```

In the custom JSP (/custom/components/menubar.jsp), remove the `<actionmenuitem>` tag named 'view_history'. Save and close the JSP page and the XML configuration file, then

restart the server. (All changes to XML files require a restart of the server, because the configuration files are read into memory the first time the configuration is required.)

Because menus are generated as JavaScript, you must clear the previous menu from the browser history (clear the cache). Result:

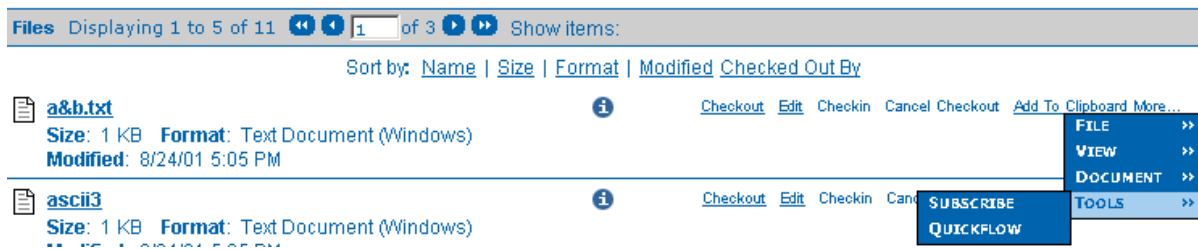
Figure 9-3. Revised view menu



Example 9-3. Changing the More... menus

In the next example, we change the menu items that are available next to each object in the folder contents view. The same action menu for dm_sysobjects is used in several drilldown pages: home cabinet, cabinets, subscriptions, my objects, and several others. Before the configuration change, objects are displayed with a menu list next to each item. The more link displays an additional set of menus as shown below:

Figure 9-4. Drilldown More... menu



In this example we will move the **Quickflow** menu item to the **Document** menu. To customize the menu items in these menus, you must extend the actionlist that is defined for dm_sysobjects.

Copy the file dm_sysobject_action.xml from /webcomponent/config/actions to /custom/config and open the file in an editor. Locate the following tag:

```
<action id="sendtodistributionlist" nlsid="MSG_SEND_TO_DISTRIBUTION"/>
```

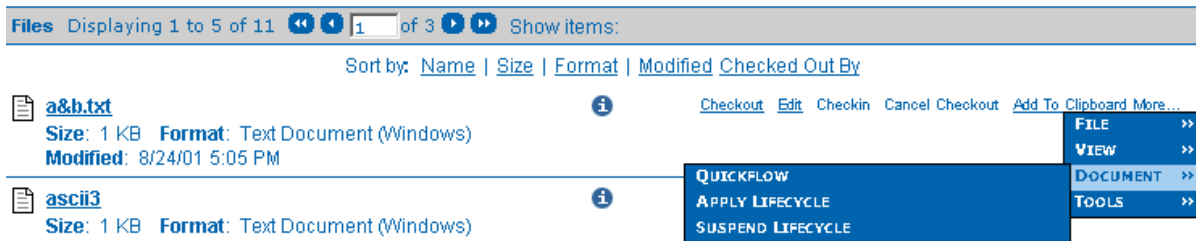
Cut and paste this tag into the beginning of the document menu, after the following tag:

```
<menu id="document" nlsid="MSG_DOCUMENT_MENU">
```

Save and close the file and then refresh the configuration files in memory by navigating to /wdk/refresh.jsp.

The resulting menu now shows the rearranged menu items:

Figure 9-5. Reconfigured More... menu



Configuring content display
















This section describes how to show and hide metadata for lists of objects in the UI using fixed columns that are specified at design time or using dynamic columns based on context-sensitive settings that are evaluated at run time.

For information on setting up data sources for a datagrid control, refer to [Configuring databound controls](#), page 184.

Example 9-4. Data display: design-time configuration

The following example uses the myfiles_streamline component in Webtop to hide and display attributes. This component has the following UI:

Figure 9-6. Webtop My Files default display

My Files		Page 1 of 2	Items per page: 10
Sort by: Name Format Size Version Modified Checked Out By			
	5.3 and SP3 Resource Plan		Checkout Checkin
	Format: Microsoft Excel Worksheet Size: 34 KB Version: 1.8 Modified: 9/14/04 10:25 AM <i>Checked Out By:</i> Lani Hardage-Vergeer		
	a&b.txt		Checkout Edit Checkin
	Format: Text Document (Windows) Size: 1 KB Version: 1.2,CURRENT Modified: 2/3/04 1:34 PM <i>Checked Out By:</i> Lani Hardage-Vergeer		
	test		Checkout Edit Checkin
	Format: Microsoft Word Document Size: 20 KB Version: 1.0,CURRENT,_NEW_ Modified: 12/16/04 9:42 AM <i>Checked Out By:</i> Lani Hardage-Vergeer		
	wdkref_controls.xml		Checkout Edit Checkin
	Format: XML Document Size: 297 KB Version: 2.4,CURRENT Modified: 2/10/05 9:39 AM		
	WDK_525_SP2_Troubleshooting.pdf		Checkout Edit Checkin
	Format: Acrobat PDF Size: 247 KB Version: 1.0,CURRENT Modified: 2/11/05 1:46 PM <i>Checked Out By:</i> Lani Hardage-Vergeer		

Copy `myobjects_drilldown_component.xml` from the `/webcomponent/config/library/myobjects` to your custom config directory. Adjust the column visibility settings to display only the object name and type, version label, lock owner, and last modification date. For example:

```
<columns>
  <column>
    <attribute>object_name</attribute>
    <label><nlsid>MSG_NAME</nlsid></label>
  </column>
  <column>
    <attribute>r_object_type</attribute>
    <label><nlsid>MSG_OBJECT_TYPE</nlsid></label>
    <visible>true</visible>
  </column>
  <column>
    <attribute>r_version_label</attribute>
    <label><nlsid>MSG_VERSION_LABEL</nlsid></label>
    <visible>true</visible>
  </column>
  <column>
    <attribute>r_modify_date</attribute>
    <label><nlsid>MSG_MODIFIED_DATE</nlsid></label>
    <visible>true</visible>
  </column>
  <column>
    <attribute>r_lock_owner</attribute>
    <label><nlsid>MSG_LOCK_OWNER</nlsid></label>
  </column>
</columns>
```

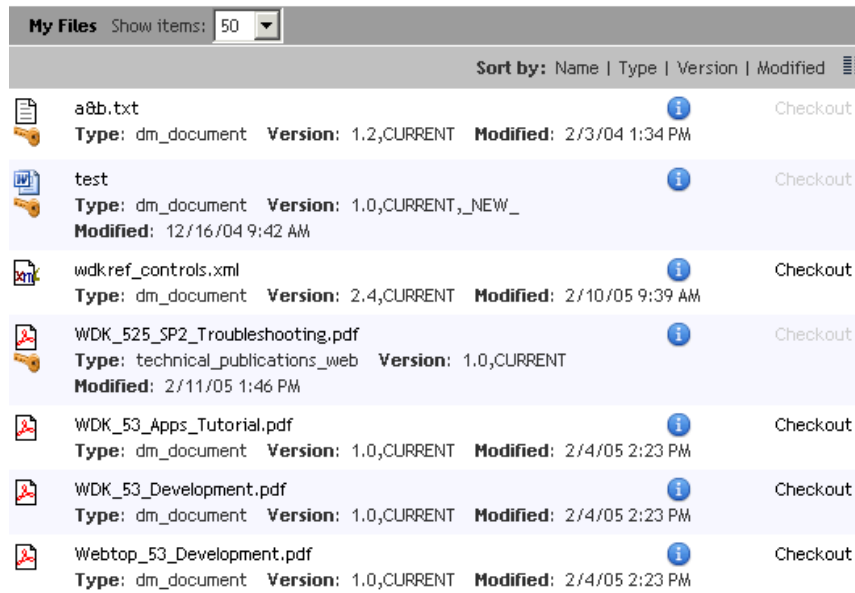
```

    <visible>true</visible>
  </column>
</columns>
...

```

The result, after refreshing the configuration files by navigating to /wdk/refresh.jsp, is shown below. (The lock owner is not displayed when the object is not locked.)

Figure 9-7. Configured My Files display



Example 9-5. Data display: Custom objects and attributes

The following example uses the myobjects_drilldown component to display custom attributes in a list of objects of mixed object type.

Copy myobjects_drilldown_component.xml from the /webcomponent/config/library/myobjects to the /custom/config directory.

Add your custom attribute columns to the list of columns that will be queried. You need a column for every attribute that will be displayed. If an object in the list is not of the custom type, no value will be displayed in the column for the custom attribute.

The following example adds two custom attributes for the custom object type technical_publications_web:

```

<columns>
  <column>
    <attribute>object_name</attribute>
    ...
  </column>
  <column>
    <attribute>tp_product_name</attribute>

```



```

        <label>Product Name</label>
        <visible>true</visible>
    </column>
</column>
    <attribute>tp_product_version</attribute>
    <label>Product Version</label>
    <visible>true</visible>
</column>
</columns>

```

Refresh the configuration files by navigating to `/wdk/refresh.jsp`, and then view My Files in the UI.

The result is the following:

Figure 9-8. Custom attributes display based on context

The screenshot shows a file management interface titled "My Files" with a "Show items: 50" dropdown. The interface has a header with "Sort by: Name | Format | Size | Modified | Checked Out By | Product Name | Product Version". Below the header, there are four file entries, each with a file icon, name, format, size, modified date, checked out by, and custom attributes (Product Name and Product Version). Action buttons (Checkout, Edit, Checkin, Cancel) are visible for each file.

File Name	Format	Size	Modified	Checked Out By	Product Name	Product Version
a8b.txt	Text Document (Windows)	1 KB	2/3/04 1:34 PM	Lani Hardage-Vergeer		
test	Microsoft Word Document	20 KB	12/16/04 9:42 AM	Lani Hardage-Vergeer		
wdkref_controls.xml	XML Document	297 KB	2/10/05 9:39 AM			
WDK_525_SP2_Troubleshooting.pdf	Acrobat PDF	247 KB	2/11/05 1:46 PM	Lani Hardage-Vergeer	WDK	5.2.5 SP2

For more information on dynamic data display through cell templates, refer to [Configuring dynamic data columns, page 232](#).

Configuring navigation base cabinet or folder

You can configure some of the navigation components to start in a specific base cabinet or folder. This limits browsing to the specific cabinet or folder and its subfolders. The doclist and drilldown components can be called with an optional `folderId` or `folderPath` argument.

The foldertree component can be called with an optional folderPath argument, but this does not limit browsing. The specified folder path is highlighted and the foldertree is expanded to display the subfolders of the specified folder. The default doclist view has the repository root as the navigation base, as illustrated in the following screen capture:

Figure 9-9. Default navigation from repository root

<input checked="" type="checkbox"/>		Name	Size	Format	Modified
<input type="checkbox"/>		ABU			8/27/99 10:30 AM
<input type="checkbox"/>		Administration			8/13/01 1:33 PM
<input type="checkbox"/>		Channels and Alliances			11/7/00 10:20 AM
<input type="checkbox"/>		Consulting			7/22/96 1:47 PM
<input type="checkbox"/>		Customer Advocacy			4/13/01 5:13 PM
<input type="checkbox"/>		Documentation Library			1/8/97 12:56 PM
<input type="checkbox"/>		documentum.com			2/15/00 1:37 PM

Example 9-6. Configuring the base cabinet in the doclist

You can configure the doclist to display any cabinet or folder as the base for navigation. The following example calls the doclist component and supplies the optional folderPath argument. The argument is given at the end of this URL:

```
http://localhost:8080/wdk51/component/doclist?folderPath=LB%20Library
```

The resulting doclist display is shown below:

Figure 9-10. Navigation from a specific folder path

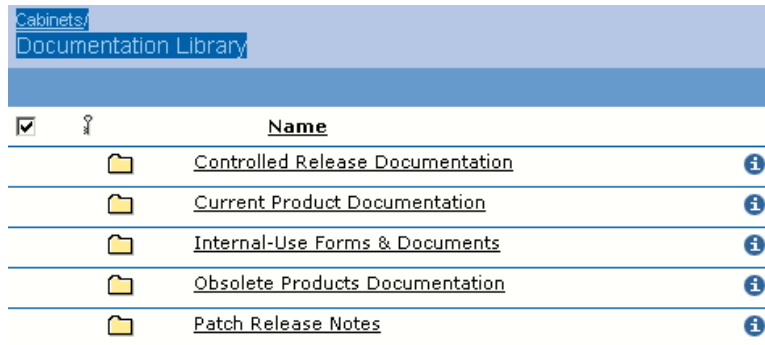
<input checked="" type="checkbox"/>		Name	Size	Format	Modified
<input type="checkbox"/>		ABU			8/27/99 10:30 AM
<input type="checkbox"/>		Administration			8/13/01 1:33 PM
<input type="checkbox"/>		Channels and Alliances			11/7/00 10:20 AM
<input type="checkbox"/>		Consulting			7/22/96 1:47 PM
<input type="checkbox"/>		Customer Advocacy			4/13/01 5:13 PM
<input type="checkbox"/>		Documentation Library			1/8/97 12:56 PM
<input type="checkbox"/>		documentum.com			2/15/00 1:37 PM

You can also supply the base location as a folder ID. For example:

```
http://localhost:8080/wdk51/component/doclist?folderId=0c00000180005680
```

When you supply the folder ID, the navigation path is displayed in the breadcrumb component, allowing the user to navigate above the base location. In the display shown below, the user can click on the Cabinets part of the breadcrumb to view all cabinets in the repository:

Figure 9-11. Navigation from a specific folder ID



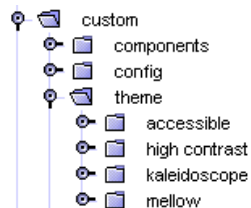
<input checked="" type="checkbox"/>		Name	
		Controlled Release Documentation	
		Current Product Documentation	
		Internal-Use Forms & Documents	
		Obsolete Products Documentation	
		Patch Release Notes	

Configuring branding

This section describes how to configure branding of your application.

Create a new branding directory in custom application directory. Copy only the themes you wish to use in your application, and create directories for your custom themes. For example, /custom/theme/accessible.

Figure 9-12. Custom theme directory



Copy the webcomponents.css file from the /webcomponent/theme/documentum/css directory to the new custom theme directory /custom/theme/accessible/css. In the style sheet, change the font sizes to larger fonts. We do not need to copy the icons and images directories, because they inherit graphics from the parent theme. Register the new theme in the app.xml file:

```

<themes>
  <theme>
    <name>accessible</name>
  </theme>
</themes>
  
```

```

    <base-theme>documentum</base-theme>
    <label><nlsid>MSG_BRAND_ACCESSIBLE</nlsid></label>
  </theme>
</themes>

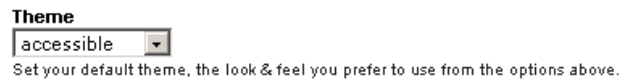
```

Create an NLS file in the /custom/strings directory that includes the BrandingServiceNlsProp.properties file from the directory /wdk/strings/com/documentum/web/common/. Add the string for your new theme to this file. (Refer to [Adding strings to properties files, page 138](#) for more information on adding strings to NLS files.) For example:

```
MSG_BRAND_ACCESSIBLE=accessible
```

Your new theme will appear as a preference selection after the J2EE server has been restarted:

Figure 9-13. New default theme



Configuring validators

This topic describes how to add or change validation for a text control or data field.

Required field — You can require the user to enter data by adding a required field validator. The following example from newCabinet.jsp validates that the user has entered a name for the new cabinet:

```

<dmf:text name="attribute_object_name" size="30"></dmf:text>
<dmf:requiredfieldvalidator name="validator"
  controtovalidate="attribute_object_name"
  nlsid="MSG_MUST_HAVE_NAME">
</dmf:requiredfieldvalidator>

```

In this example, the text control is named "attribute_object_name". This name is used as the value for the validation control controtovalidate attribute value. The error message is specified as the value of the nlsid attribute, which can be an externalized string, as in the example, or a hard-coded string.

Comparison validator — The comparison validator checks the value of one control to the value of another. The following example compares the passwords entered in two fields:

```
<dmf:password id='ChangePassword' name='ChangePassword' size='40'
```

```

    defaulttonenter='true' />
<dmf:password id='ConfirmPassword' name='ConfirmPassword' size='40'
    defaulttonenter='true' />
<dmf:comparevalidator name="validator" controltovalidate="ConfirmPassword"
    type="string" operator="equal" comparecontrol="ChangePassword"
    nlsid="MSG_CONFIRMPASSWORD_NOTEQUAL" />

```

Input mask — You can apply the input mask validator to verify that the user’s input matches an expected pattern. In the following example, the input mask rejects ZIP codes that do not match the mask pattern:

```

<dmf:text name='zip' size='10' defaulttonenter='true' />
<dmf:inputmaskvalidator name="zipval" controltovalidate="zip"
    inputmask="#####" enabled='true' errormessage="Must have 5 digits" />

```

Validating repository attributes — The `docbaseattributevalidator` tag is generated by the `docbaseattribute` tag, so you do not have to add it explicitly to validate repository attribute values. Any value that is entered into a `docbaseattribute` tag field is automatically validated by `inputmaskvalidator`, `multivaluesinputmaskvalidator`, `requiredfieldvalidator`, and `varequiredfieldvalidator`.

The `docbaseattribute` tag generates a table that displays an attribute. The attribute can be displayed as readonly or editable, based on the value of the `readonly` attribute. The following example from `checkin.jsp` adds the `docbaseobject` tag, which gets information about the object from the repository. The `docbaseattributevalue` tag displays the value of the `object_name` attribute. The `docbaseattribute` tag displays the format:

```

<dmfx:docbaseobject name="object" />
<table border="0" cellpadding="2" cellspacing="0">
<tr>
    <td>
        <dmfx:docbaseattributevalue object="object" attribute="object_name"
            readonly="true" />
    </td>
</tr>
<tr>
    <td>
        <dmfx:docbaseattribute object="object" attribute="a_content_type"
            readonly="true" />
    </td>
</tr>
</table>

```

Configuring application startup

This topic describes typical configuration of the startup settings for an application. You can configure startup settings for the root Web application and for the application layers within WDK.

Your Web application contains at least two application layers: WDK and webcomponent. It can also contain the Webtop application layer, and it contains one or more custom application layers that you have created. You can configure application startup settings in the following files:

- `/WEB-INF/web.xml`: Configures the root application as well as application startup context parameters used by WDK and, optionally, startup parameters used by your application classes.
- `/application_root/app.xml`: Configures the following settings in an application layer (WDK, webcomponent, Webtop, and your custom application):
 - Supported locales: For information on configuring locale support, refer to [Configuring and localizing strings, page 137](#)
 - Default repository and domain
 - Branding resource directories: For information on configuring branding, refer to [Branding an application, page 122](#).
 - Accessibility features: For information on accessibility features in WDK, refer to [Accessibility service, page 584](#).
 - Content transfer default directories, buffer size, and debugging: For information on configuring content transfer settings, refer to [Application configuration file \(app.xml\), page 58](#).
 - Webtop display: You can configure the default number of items that will be displayed in the Webtop classic view and in the streamline cabinets, folders, and files views.

Configuring accessibility

The accessibility settings are contained within the `<accessibility>` element. You can configure the following behavior:

alttextenabled — Generates an HTML alt attribute for every image. The text for alt attributes is specified in a properties file in `/WEB-INF/classes/com/documentum/web/accessibility/icons` or `/images` directory. The lookup key is the file name. For example, `FormatAltNlsProp.properties` contains a lookup key for repository format icons. The icon `f_aiff_16.gif` has an alt text of AIFF sound.

Note: The alt text feature overrides tooltips on images. If alt text does not exist for an image, no tooltip will be generated. You should specify an empty alt string for images that should not be picked up by reader software, for example:

```
<img src='<%=Form.makeUrl(request, "/wdk/images/space.gif")%>'
      width="1" height="1" alt="">
```

keyboardnavigationenabled — Enables navigation using the keyboard. By default keyboard access to menus, buttons and tabs are disabled unless the accessibility option is on.

shortcutnavigationenabled — Generates shortcuts to the top and bottom of a tree.

For more information on accessibility features in WDK, refer to [Accessibility service, page 584](#).

Configuring the properties container

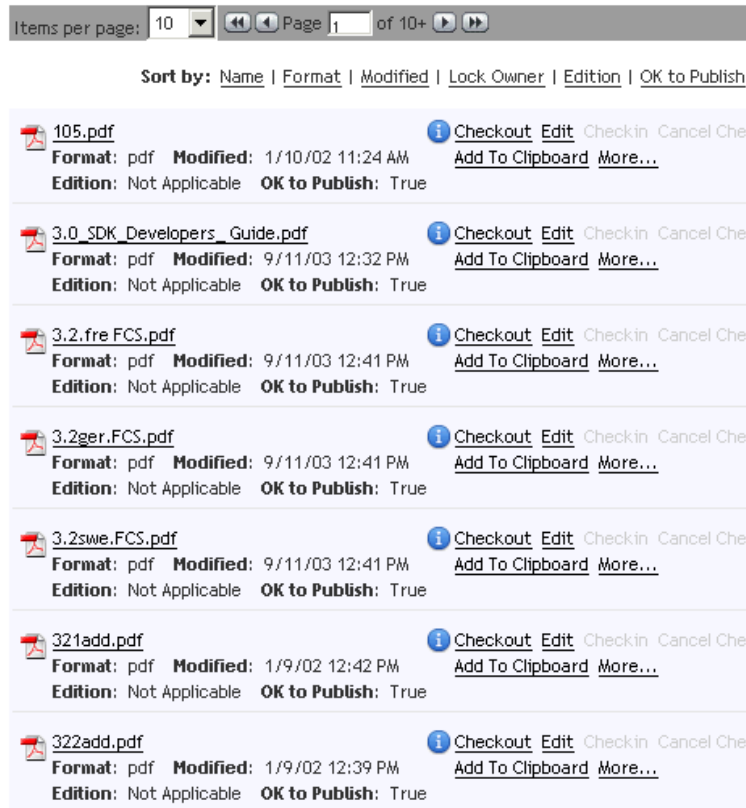
This topic describes customization of a properties page. The properties component is a container for several types of properties files: attributes, permissions, and object history (audit trail). The following example adds the locations component to the list of properties for an object.

Copy the properties component configuration file from `/webcomponent/config/library/properties` to your custom config directory. Add the locations component to the list of contained components:

```
<contains>
  <component>attributes</component>
  <component>permissions</component>
  <component>history</component>
  <component>locations</component>
</contains>
```

The result:

Figure 9-14. Adding a component to the properties container



Configuring attributes

You can configure the following features of the attributes component definition:

- Layout pages: You can change the <start> and <all> JSP pages to use customized JSP pages in your custom application directory.
- Filter: Change the display for different repository roles
- Read only: Make the attributes read-only or editable
- Change the ID of the help page that will be displayed
- Attribute list: Change the attributelist that is used to configure the display of attributes from the data dictionary

You can display attributes for custom object types or other contexts using docbaseattributelist configuration files. For more information, refer to [Displaying and validating attributes](#), page 192.

- `Docbaseobjectconfiguration`: Change the configuration for this control to use your custom tag or formatting classes to render specific attributes or attribute types.

For more information on this customization feature, refer to [Modifying the display and handling of attributes](#), page 395.

The following topics provide examples of configuring attributes:

- [Configuring attribute layout](#), page 313
- [Configuring an attribute list](#), page 314

Configuring attribute layout

You can add attribute controls to a JSP page to have layout control. You can add a `docbaseattributelist` control to the page to have your attributes displayed generated from the data dictionary. For more information, refer to [Displaying and validating attributes](#), page 192.

You can format individual attribute controls by placing a formatter control around the attribute control, for example, a `stringlengthformatter` or a `booleanformatter`. Refer to the WDK and Webtop JSP pages for examples of formatter controls around attribute controls.

To use customized JSP pages, place the customized pages in the `/custom` directory where they will not be overwritten by WDK upgrades. Specify the `<start>` page for your custom attribute component. The following example extends the webcomponent `sysobject` attributes component and substitutes customized JSP pages:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<config version='1.0'>
<scope type='dm_sysobject'>
<component id="attributes" extends="attributes:webcomponent/
  config/library/attributes/attributes_dm_sysobject_component.xml">
  <params>
    <param name="objectId" required="true"></param>
    <param name="readOnly" required="false"></param>
    <param name="enableShowAll" required="false"></param>
  </params>
  <pages>
    <start>/custom/research/isotopes/attributes.jsp</start>
    <all>/custom/research/isotopes/attributes_all.jsp</all>
  </pages>
</component>
</scope>
</config>
```

Configuring an attribute list

To display a set of custom attributes in a single component such as checkin, you must create an XML attributelist configuration file containing an <attributelist> element that is scoped to your custom object type. Each component can configure the list of attributes that are displayed, and within the component you can configure a different list of attributes based on object type.

You can set up the attribute list using Documentum Application Builder (DAB). Create a list of attributes to be displayed for each scope that will have a different display. For example, if you wish to display a list of custom attributes for the object type technical_publications_web, set up the technical_publications_web scope type in DAB. You can also set these attributes in categories, which will be displayed as tabs of properties in the UI.


Example 9-7. Removing attributes from display (data-dictionary list of attributes)

The WDK checkin component uses the resource file checkin_docbaseattributelist.xml, located in /webcomponent/config/library, to specify the attributes that will be displayed in the checkin UI. The <attributelist> element ID is "checkin", which matches the value of the attrconfigid attribute in the <dmfx:docbaseattributelist> tag of the checkin JSP page . (Refer to [Displaying and validating attributes, page 192](#) for more details.)

To change the attributes that are displayed on checkin, you must create an attributelist definition for your custom type.

Before you customize checkin, the following attributes are displayed on checkin of a custom type as shown in the screen capture below:

Figure 9-15. Default checkin attributes for a custom type

 testwebdoc

Version 1.0, CURRENT
Type technical_publications_web
Format

Save as: 1.0 (same version)
 1.1 (minor version)
 2.0 (major version)

Version label:

Description:

Format: Unknown

Abstract: *

Document Type: [Edit](#) *

Parent Edition: [Edit](#) *

Product Name: *

Product Version: [Edit](#) *

Applicable OS: [Edit](#) *

Applicable RDBMS: [Edit](#) *

Language: *

Locales: [Edit](#) *

OK To Display?:

Date Published: Date Hour Minute Second

Doc Part Number:

Effective Date (TS): Date Hour Minute Second

Expire Date (TS): Date Hour Minute Second

Obsolete Date (TS): Date Hour Minute Second

[+] [Show options](#)

The checkin attributelist configuration file, checkin_docbaseattributelist.xml, is located in /webcomponent/config/library. Copy this configuration file to /custom/config and name it with information about the custom type, for example, checkin_tpw_docbaseattributelist.xml.

Open the file and locate the following line:

```
<scope type="dm_document">
```

Remove the entire contents of this element, from <scope type="dm_document"> to </scope>. There are two scope elements in this file, and we don't need the dm_document

scope. The definition for `dm_documents` will be inherited, and we must define only the scope for the custom type.

In the next `<scope>` element, add the custom type:

```
<scope type="technical_publications_web">
```


Make sure that the value of the element `<data_dictionary_population>.<enable>` is true so that the attribute list will be read from the repository data dictionary. The scope type will be matched to a scope list in the repository, as you configured it using DAB. If your repository is pre-5.2, refer to [Displaying and validating attributes, page 192](#) for information on setting up attribute lists in the configuration file.

For checkin, we want to remove certain attributes that should not be changed after the document has been created or imported: document type, parent edition, original language of the document, publication date, and doc part number. We will add those attributes to the `<ignore_attributes>` in the definition:

```
<attribute name="tp_literature_type"/>
<attribute name="tp_edition"/>
<attribute name="tp_language"/>
<attribute name="tp_modify_date"/>
<attribute name="tp_part_number"/>
```

Save and close the XML file, and then refresh the configuration files in memory by navigating to `/virtual_root/wdk/refresh.jsp`. The final result:

Figure 9-16. Custom checkin attributes for a custom type

 testwebdoc

Version 1.0, CURRENT
Type technical_publications_web
Format

Save as: 1.0 (same version)
 1.1 (minor version)
 2.0 (major version)

Version label:

Description:

Format: Unknown

Abstract: *

Product Name: *

Product Version: [Edit](#) *

Applicable OS: [Edit](#) *

Applicable RDBMS: [Edit](#) *

Locales: [Edit](#) *

OK To Display?:

Effective Date (TS): , : :

Expire Date (TS): , : :

Obsolete Date (TS): , : :

Creating a custom object filter

Several components present the user with filters to display certain types of objects: files, folders, dm_document objects, or all objects. This example adds a filter that will display objects of a custom type. The custom type in this example is named technical_publications_web, but any custom type can be substituted to create a filter.

Note: If the custom type does not exist in the repository, the dropdown list of filters will not be rendered at all. To use the custom filter in all your repositories, make sure the type is installed in each repository even if no objects of that type will be stored in that repository.

Object filters are available in several components. Add your custom filter to every component that can display your custom object type. This example adds the custom filter to the Webtop objectlist component. (In WDK without Webtop, you could add

the filter to the doclist component.) Copy the objectlist component configuration file `objectlist_component.xml` from `/webtop/config/` to `/custom/config/`. Open the file and make the following changes:

1. Change the component definition to extend the Webtop component:

```
<component id="objectlist" extends="objectlist:
webtop/config/objectlist_component.xml">
```

2. Copy the `objectfilters` element from `/webcomponent/config/navigation/doclist/doclist_component.xml` into your definition. You must do this because you are going to add an `<objectfilter>` element, and the filters are inherited in the Webtop objectlist component from the parent doclist component:

```
<objectfilters>
  <objectfilter>
    <label><nlsid>MSG_FILTER_FILES_FOLDERS</nlsid></label>
    <showfolders>true</showfolders>
    <type>dm_document</type>
  </objectfilter>

  <objectfilter>
    <label><nlsid>MSG_FILTER_FILES</nlsid></label>
    <showfolders>false</showfolders>
    <type>dm_document</type>
  </objectfilter>

  <objectfilter>
    <label><nlsid>MSG_FILTER_FOLDERS</nlsid></label>
    <showfolders>true</showfolders>
    <type></type>
  </objectfilter>

  <objectfilter>
    <label><nlsid>MSG_FILTER_ALL</nlsid></label>
    <showfolders>true</showfolders>
    <type>dm_sysobject</type>
    <showallversions>true</showallversions>
  </objectfilter>
</objectfilters>
```

3. Before the last `<objectfilter>` element, add your custom filter. Substitute the custom object type for the value of the `<type>` element:

```
<objectfilter>
  <label>Show Web Documents</label>
  <showfolders>false</showfolders>
  <type>technical_publications_web</type>
</objectfilter>
```

4. Refresh the cached configurations by navigating to `/wdk/refresh.jsp`, and then view a cabinet or folder that contains your custom type. (Remember that you have not added your filter to the homecabinet components, so the filter will not be available in those views.)

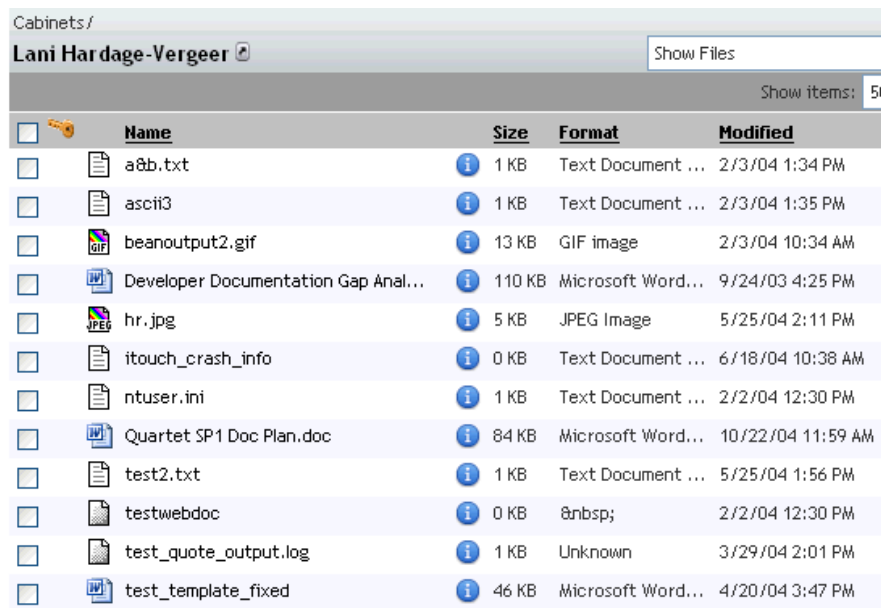
The results of the custom filter will be similar to the following:

Figure 9-17. Object list with custom filter



When you apply the files filter, which resolves to dm_document, a different set of objects is displayed:

Figure 9-18. Object list with standard files filter



Customizing WDK Applications

This section of the development guide describes the customization methodology for WDK-based Web applications. The following chapters describe the WDK framework and customization:

- [Chapter 10, Development Environment and Tools](#)
- [Chapter 11, Component, Action, and Control Design Guidelines](#)
- [Chapter 12, Customizing Controls](#)
- [Chapter 13, Customizing Components](#)
- [Chapter 14, Using the Configuration Service](#)
- [Chapter 15, Customizing actions](#)
- [Chapter 16, Customizing Roles](#)
- [Chapter 17, Customizing Content Transfer](#)
- [Chapter 18, Customizing Authentication](#)
- [Chapter 19, Managing Sessions](#)
- [Chapter 20, Customizing Search](#)
- [Chapter 21, Implementing Component and User Preferences](#)
- [Chapter 22, Other Customizations](#)
- [Chapter 23, Using Business Objects in WDK](#)
- [Chapter 24, Customization Examples](#)

Development Environment and Tools

The following topics describe setup of the development environment and tools you can use in developing WDK-based applications:

- [Using an IDE, page 323](#)
- [Troubleshooting WDK-based applications, page 324](#)
- [Runtime errors, page 324](#)
- [Tracing, page 333](#)
- [Logging, page 343](#)
- [Performance, page 344](#)
- [Finding component information, page 353](#)
- [Comment stripper, page 353](#)
- [Testing components, page 354](#)
- [Debugging tips, page 354](#)

Refer to [Performance, page 344](#) and [Chapter 11, Component, Action, and Control Design Guidelines](#).

Using an IDE

If you develop a Web application using an integrated development environment (IDE), you must configure WDK to run within that IDE. The documentation for your integrated development environment (IDE) describes how to set the classpath for your Web application. You must set the classpath to include WDK libraries in order to run or compile WDK-based applications from within your IDE.

The Documentum Java libraries must be referenced in the J2EE server classpath because they are outside of the Web application. The home directory must be referenced in the J2EE server path because it contains native libraries. The installers for WDK and its client applications set the J2EE server classpath and the path to the Documentum home directory when you run the installer on the J2EE server host. Refer to *Web Development Kit and Applications Installation Guide* for more information.

If your Java IDE does not include `j2ee.jar` (or some subset of it) in its library directory, you must install it on your local system and reference it in your IDE classpath. You must also reference all of the jar files that are installed by the WDK installer to your `DOCUMENTUM_HOME` directory (default=`C:\Program Files\Documentum\shared`).

Consult the documentation for your IDE for instructions on how to set up a deployed Web application for development, debugging, and compilation. The tutorial *Web Development Kit and Applications Tutorial* describes how to set up NetBeans, a free J2EE IDE, to work with WDK.

Troubleshooting WDK-based applications

The Documentum Web Development Kit provides the following troubleshooting tools:

- **Tracing flags:** Trace the following types of operations or content manipulation: sessions, JSP requests, locales, actions, configuration, roles, preferences, resources, clipboard, controls, control tags, form navigation and history, validation, repository attributes, content transfer, components, containers, and WDK 4 components. For the full list of tracing flags and their usage, refer to [Tracing, page 333](#).
- **Logging:** Use the open source Apache log4j tool. For more information on logging, refer to [Logging, page 343](#).
- **Testing components:** Test controls and components using the test pages and components in the `/wdk/samples` directory. The file `wdk/config/txtest_component.xml` defines simple components that test the controls. The component definitions specify the start JSP page that is located in `/wdk/samples`.

Refer to *Documentum Web Development Kit Release Notes* for a list of bugs and workarounds. You may have encountered a known issue that has a workaround.

Runtime errors

The following topics describe errors that are encountered during connection to a deployed WDK-based application. Some general tips to try when you encounter runtime errors are the following:

- **Clear the browser cache.** The browser caches JavaScript even when you have set your browser to refresh a URL on every visit.
- **Delete generated class files for JSP pages.** When you make changes to JSP pages that are included in another JSP page, the application server does not detect those changes. When you make changes to Java classes or Java properties files that are called by JSP pages, the application server does not detect such changes.

- Refresh the application XML configuration files and data dictionary when you make changes to them. Visit `/wdk/refresh.jsp` to refresh the configurations and data dictionary in memory.

Error loading main component

The following error is displayed when the WDK-based application is first loaded:

```
This <main> component is invoked when the root application is loaded and when timeout ar
```

Cause: Your custom `\app.xml` extends `webcomponent\app.xml`. If you change it to extend `webtop\app.xml`, or the top application layer in a Webtop-based application, the error may be resolved.

Show All Properties does not work

The following issues can result in failure of the Show All Properties feature:

- When the application server locale does not match the locales in the repository, the Show All Properties feature does not work. For example, if you have an application server on an English machine locale (`en`) and the repository has only one installed locale, Japanese. The query attempts to retrieve information for the repository locale

Solution: The repository must publish the data dictionary for the locale of the application server.

- When a value assistance query for a custom type contains an error, the error message `DM_API_E_BADID` is displayed.

Solution: Check the query syntax using the `dql` component.

Properties do not display after data dictionary change

After changes to the data dictionary, the properties no longer display in the WDK application. This is due to object type information being cached in the DMCL layer on the application server host.

Workaround: Delete the DMCL cache. In Tomcat, the object type information is cached in `TOMCAT_HOME\bin\dmcl\object_caches`. In Weblogic the directory is `WEBLOGIC_HOME\user_projects\domain_name \dmcl\object_caches`. Delete this folder and restart the application server.

WebLogic compiler fails

The WebLogic compiler has a limitation on the number of JSP tags that can be processed in a JSP page. There is no single number that reveals this limitation; it is a combination of the number of tags and the number of attributes that are set on those tags.

If the compiler fails on a custom JSP page that has a large number of JSP tags, use the following workaround: Move some of the tags into a separate JSP page that you include with a JSP:include directive. For example, the Documentum Administrator menubar JSP page moves individual menus into separate pages, as follows:

```
<tr>
  <td> </td>
  <!-- Menus -->
  <jsp:include page="file_menu.jsp"/>
  <jsp:include page="edit_menu.jsp"/>
  <jsp:include page="view_menu.jsp"/>
  <jsp:include page="tools_menu.jsp"/>
</tr>
```

A sample of one of the included pages follows:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ page errorPage="/wdk/errorhandler.jsp" %>
<%@ taglib uri="/WEB-INF/tlds/dmform_1_0.tld" prefix="dmf" %>
<%@ taglib uri="/WEB-INF/tlds/dmformext_1_0.tld" prefix="dmfx" %>
<%@ page import="com.documentum.web.form.Form" %>
<%@ page import="java.util.Enumeration" %>
<%@ page import="com.documentum.web.form.Control" %>
<%@ page import="com.documentum.web.common.ArgumentList" %>

<td>
  <dmf:menu name="tools_menu" nlsid="MSG_TOOLS" width="50">
    <dmf:menu name="doc_lifecycle" nlsid="MSG_LIFECYCLE">
      <dmfx:actionmenuitem dynamic="multiselect"
        name="doc_promotelifecycle" nlsid="MSG_PROMOTE_LIFECYCLE"
        action="promote" showifinvalid="true"/>
      ...
    </dmf:menu>
  </dmf:menu>
</td>
```

WebLogic slows, throws exceptions, or crashes

WDK applications do not currently support running in Archive mode. BEA WebLogic can slow, throw exceptions, or crash with an application that has a large number of JSP pages. BEA engineers have recommended setting the MaxPermSize to 128.

(WebLogic) java.io.IOException: Not enough space

This error message is displayed because of one of three conditions: not enough RAM, not enough file descriptors, or not using native threading. BEA suggests the following workarounds:

- Add more RAM if you have only 256 MB.
- Raise the file descriptor limit, for example: set rlim_fd_max = 4096 set rlim_fd_cur = 1024
- Use the -native flag to use native threads when starting the JVM.

Future dates do not display correctly

Dates do not display correctly if the year is more than 20 years beyond the current date. The date will be displayed as being in the 20th century. For example, if the user enters a date value of 07/30/2033 and views the date again, the date is displayed as 7/301933.

Solution: The repository server.ini file must be modified to send a four-digit year date. In the [SERVER_STARTUP] section of the server.ini file, set the enforce_four_digit_year flag to true and then restart Content Server. You must also restart the J2EE application server for the Web application.

JavaScript error on application connection

When you log into an installation of a WDK application, you may see the following JavaScript error message in the browser:

```
...Error: Object doesn't support this property or method
```

This error can be caused by IE 5.5 SP1, which is not a supported browser. The error is also caused by using IE without a Java virtual machine. You must upgrade to a supported version of the browser and install a supported VM.

Error "Configuration base has not been established"

This error is almost always due to errors with _dmfRequestId values. The value may be old (used in a prior URL), null or from the wrong client number (frame). Do not hard code dmfRequestId values in URLs.

Hard-coded URLs in test scripts can also generate a null pointer exception after `FormProcessor.invokeMethod()` is called.

Application no longer starts after code change

After making code changes to your application, you may see the following error message in the error stack trace:

```
Component Definition main 'component[id=main]' does not exist
  within context REQUEST()SESSION()APP() :
```

You must delete compiled JSP class files and restart the application server. (All JSP pages are compiled in class files by the application server at the time the page is requested. Refer to your application server documentation to find the location of the JSP class files.)

(Tomcat) Application slows down

Tomcat can slow down even when the system has a lot of memory available. The default maximum memory size for the JVM is 64 MB. The application slows down when it reaches the memory ceiling and begins swapping objects in and out of the base memory.

Solution: Increase the Java memory heap by adding the following command line to the Tomcat server startup command file (`catalina.bat`):

```
set JAVA_OPTS =-Xms60m -Xmx300x
```

This setting will increase the minimum JVM memory from 2 MB to 60 MB and the maximum memory from 64 MB to 3000 MB.

Page not found errors in if HTTP 1.1 not enabled in client browser

The browser reports Page not Found when the user clicks on a cabinet or folder. You must enable HTTP 1.1 in the IE browser: Go to the **Tools** menu and select **Internet options**. On the **Advanced** tab, in the HTTP 1.1 settings section, check **Use HTTP 1.1**.

DFC business object no longer works

The WDK and WDK client installers replace the business object registration file `dbor.properties`. The installers make a backup of the file in `DOCUMENTUM_HOME/User_dir/config`, for example, `dbor.properties.webtop.backup`.

Solution: Add your custom entries from the backed up `dbor.properties` to the current `dbor.properties` and then restart your DFC-based applications.

Application runs out of sessions

There are several reasons that can cause an application to run out of sessions:

- The application server may have a limited number of sessions available. Some demo or developer versions limit sessions to 2 or 3.
- The application server machine's `dmcl.ini` file does not provide enough settings. The setting `max_session_count` in the section `[DMAPI_CONFIGURATION]` should have a value of at least 1000. You can set the count higher if necessary:

```
max_session_count=1000
```

- DMCL session pooling is turned off. If session pooling is turned on at the DMCL level, the session manager releases sessions 5 seconds after a client releases the `DfSession` object. When session pooling is turned off, the session is not released for several minutes.

Refer to the Content Server documentation for instructions on turning on session pooling.

Browser navigation renders actions or links invalid

Any HTML element or JSP tag that triggers a user event, such as a button, link, action button or action link, must be named in order for it to be properly retrieved in the browser history. If the element is not named, it will not be placed in the history snapshot and will not work properly when the user returns to the form. In the following example, the `actionlink` has a name, which ensures that it will be called when the user returns to the page:

```
<dmfx:actionlink name="viewlnk" showifdisabled="true" showifinvalid="true"
  visible="true" datafield="object_name" action="view">
  <dmf:argument name="objectId" datafield="r_object_id"/>
</dmfx:actionlink>
```

Content transfer fails

Content transfer operation fails with an error message to the console and log. There are several known causes for this failure:

1. The error message is generated (substitute the current content transfer operation for *operation_name*):

```
operation_name fails
```

The application server host must have a temporary content transfer directory that matches the directory specified in the /wdk/app.xml file element <server>.<contentlocationunix> or <server>.<contentlocationwindows>. The user who owns the application server instance must have write permissions on this temp directory. This directory is used for temporary transfers between the client and the application server and between the application server and the repository. The temporary files are cleaned up after the content transfer operation has completed.

On UNIX servers in which the app server instance owner does not have write permissions, the error message is the following:

```
ERROR: Failed to download document for viewing...  
NotSerializableException: com.documentum.web.form.FormHistory
```

2. The registry entry HKEY_Local_Machine/Software/Documentum/Common/CheckoutDirectory does not exist on a machine before performing a Content Transfer Operation. The following error is generated:

```
com.documentum.web.contentxfer.applet.registry.  
RegKeyException: Registry Error...
```

The administrator can run a script to create this key on all user machines.

3. The Microsoft browser JVM has become damaged, perhaps by a Sun SDK installation, and must be reinstalled.

Contact Microsoft for a browser JVM installer.

4. The Sun plugin is not recognized by the browser.

Certain versions of the Sun plugin are not recognized by the browser due to an error in the plugin. Uninstall the plugin and install an older or newer version.

(Windows) Applet installation fails on client

Symptom: (Client) The IE browser can't install the WDK content transfer applets, with the message "Can't find the InstallApplet class".

Cause: Some versions of IE do not have a Java Virtual Machine (JVM) installed.

Solution: You must install either the Sun JVM plugin or the Microsoft JVM. (The Sun plugin is required for UCF content transfer and user preferences. The MS VM can be used for HTTP content transfer.)

To enable the Java plugin in IE

1. On the client, ensure that you have installed the Sun Java plugin.
2. Open Internet Explorer
3. On the Tools menu, select Internet Options.
4. Select the Advanced tab.
5. Find the Java (Sun) section and check **Use Java 2 vxxx for <applet>** where xxx is a version number.
6. In the Microsoft VM section, make sure all check boxes are unchecked
7. Close all open IE windows and restart IE.

Cannot import an XML file

There are several known causes for this problem.

1. If the application is using HTTP content transfer, XML files can be imported only with the default XML application. Entities (descendants) are not imported.
2. If an XML file has been encoded in an encoding other than UTF-8, and the file includes non-ASCII characters, the import will fail with the Microsoft browser JVM.
Encode XML files in UTF-8, or use the Sun Java plugin for all users. Refer to [\(Windows\) Applet installation fails on client, page 330](#) for information on changing the browser JVM on the client and in the server setting.
3. If the XML application DTD does not have a .dtd extension, a null pointer exception is displayed. Rename the DTD file with a .dtd extension.
4. In Netscape/Windows, when the user invokes content transfer on an XML document, the full applets (containing an XML parser) are installed and the user is prompted to restart the browser. When the user restarts the browser and attempts a content transfer operation, there is a security error (cannot load DLL) or null pointer exception.

Even though the browser is closed, Netscape may be running a process in the background. Close all Netscape processes in the Windows task manager and restart the browser. Refer to your system administrator for information on how to close hidden processes.

Cannot check in XML file

When the user attempts to check in an XML file, the error returned is the following:

```
Exception: java.io.UnsupportedEncodingException: UTF-8 [Could not load class:  
sun.io.CharToByteUTF-8]  
java.lang.UnsatisfiedLinkError: java/security/AccessController.doPrivileged
```

Cause: This error is seen when the user has the Microsoft VM in the browser and has JAVA_HOME/jre/lib/rt.jar from the Sun Java SDK on the system classpath. The browser VM picks up java.security.AccessController from Sun instead of its own.

Solution: Remove rt.jar from the system classpath or switch the browser to use the Sun Java plugin.

java.lang.verify error in WDK application after installing another Documentum product

If you install a Documentum product with a version of 4.x, that product may install its own DFC 4. When that product is launched, DFC 4 is loaded into memory, and WDK-based applications will not work properly.

Unable to locate checked out objects after installing WDK-based application

If your client hosts have checked out objects using DFC 4.2 or lower, they will not be located by DFC 5. Documentum began using the HKEY_CURRENT_USER registry hive recommended by Microsoft with version 4.3 of Documentum products.

If your clients have objects still checked out when they connect to a WDK 5 client application, they can manually check in those objects by selecting Checkin from File in the checkin component. It is recommended, however, that clients check in all objects before connecting to a WDK 5 client application.

(WebLogic) Invalid ticket (content transfer fails)

Symptom: Error message during content transfer:

```
Invalid ticket. Content file could not be found.
```

Solution: Check the HTTP keep alives setting on the WebLogic server. The setting must be on.

Controls don't display any repository data

All controls in the dmfx tag library require a Documentum session, which is obtained in the component class. Any tag in the dmf tag library that sets a value based on a datafield or specifies a query to populate the control also requires a Documentum session. Make sure that the component whose JSP page contains the control is getting a session.

Tracing

You can use the WDK tracing flags in your JSP pages and Java classes, and you can add your own tracing flags to trace operations that are used in more than one class. After you enable tracing, tracing statements will be written to the wdk.log file in your DOCUMENTUM_HOME/config directory. (You selected a DOCUMENTUM_HOME directory when you first installed a Documentum product.)

Tracing has a more general usage than logging. Logging is generally used for debugging within a class or for creating an audit log.

Tracing is described in the following topics:

- [Turning on WDK tracing, page 333](#)
- [Using DFC tracing, page 334](#)
- [Using DMCL tracing, page 334](#)
- [WDK tracing flags, page 335](#)
- [Adding custom tracing flags, page 342](#)
- [Client-side tracing, page 342](#)

Turning on WDK tracing

You must enable tracing for the current session by navigating to /wdk/tracing.jsp and checking the box that enables tracing. You can enable tracing for all sessions by setting SESSIONENABLEDBYDEFAULT to true in /WEB-INF/classes/com/documentum/debug/TraceProp.properties.

There are several ways to turn on a particular tracing flag in your WDK application:

- Set the appropriate tracing flag to true in `TraceProp.properties`. The trace flags are read by the Trace when a Java class sets trace strings.
- Navigate to the tracing JSP page in the application. For WDK applications and Webtop, `tracing.jsp` is located in `/wdk`. When you set a trace flag through the tracing JSP page, the trace value overrides the value in `TraceProp.properties`. For Web Publisher, the tracing page `tracing.jsp` is located in `/wp/app`.
- Navigate to the tracing page in Web Publisher.

For more information on Web Publisher tracing flags, refer to *Web Publisher Development Guide*.

Using DFC tracing

To turn on DFC tracing, set the following preference in your `dfc.properties` file:

```
dfc.resources.diagnostics.enabled=true
```



Caution: Do not store `IDfSession` objects as member variables. The session may time out and cause a runtime error. Instead, every time a session is needed in that class, call the Component class `getDfSession()` method. `IDfTypedObjects` obtained through `IDfCollection` do not cause a problem. (They are a memory-cached row from a collection.)

Using DMCL tracing

You can trace native library calls on the J2EE server host through DMCL tracing. To turn on or off DMCL tracing in WDK or Webtop, navigate to the tracing JSP page, substituting the actual server name, port number, application name, and path to your log file in the URL:

```
http://my_server:portnumber/app_name/wdk/dmclTrace.jsp?level=10&logfile=c:\dmcl.log
```

In this example, the log file will be found at `C:\Program Files\Apache Software Foundation\Tomcat 5.0\bin`.

After you turn on DMCL tracing, you will see a message indicating that DMCL trace was called. Click the browser button to get back into your application, and then click the Refresh button.

You can turn on DMCL tracing by adding the following lines to the `dmcl.ini` file on the application server, in the `[DMAPI_CONFIGURATION]` section:

```
trace_file=path_to_log_file
```

```
trace_level=trace_level
```

For example:

```
trace_file=c:\dmcl.log  
trace_level=10
```

The dmcl trace log will be located in the location specified in the trace command. If no location is specified, it will be in the current directory of the traced program, in a file called api.log, for example, webtop/WEB-INF/dmcl.log.

Trace levels are cumulative. Level 0 is turned off, levels 1–4 are server trace levels, 9 is API calls, 10 is timing trace. The timing trace is useful for finding queries that need optimization.

The log levels for DMCL tracing are described for the Tracing API in *Content Server API Reference Manual*.

WDK tracing flags

WDK tracing flags are enumerated in the WDK resource file TraceProp.properties located in /WEB-INF/classes/com/documentum/debug. This file contains all tracing flags that are defined in your application. If there is an unknown flag in this file, the Trace class initialization will generate a warning message but will continue.

Note: You must enable tracing for the current session using one of the following methods:

- Set the SESSION flag (mandatory) and another other flags you require in TraceProp.properties and then restart the application server.
- Use a browser to navigate to /wdk/tracing.jsp and check the box that enables tracing.

You can enable tracing for all sessions for setting SESSIONENABLEDBYDEFAULT to true in /WEB-INF/classes/com/documentum/debug/TraceProp.properties.

Some WDK client applications such as Web Publisher add their own tracing flags. The Web Publisher tracing flags are located in the file WpTraceProp.properties in /WEB-INF/classes/com/documentum/wp/resources.

Tracing flags are described in the following topics:

- [Tracing sessions, page 336](#)
- [Tracing WDK framework operations, page 337](#)
- [Tracing controls and validation, page 337](#)
- [Tracing JSP processing, page 338](#)
- [Tracing components and applications, page 339](#)
- [Tracing virtual links, page 339](#)
- [Tracing servlets, page 340](#)

- [Tracing asynchronous operations, page 340](#)
- [Tracing content transfer, page 341](#)

Tracing sessions

The following tracing flags in `com.documentum.web.common.Trace` and `web.formext.Trace` can be used to trace HTTP and Docbase sessions:

Flag type	Description
SESSIONSTATE	Traces changes to HTTP session object and attributes
SESSIONSYNC	Traces session locking and unlocking
SESSIONENABLEDBY-DEFAULT	Traces all sessions when set to true;
SESSIONTIMEOUT-CONTROL	Traces changes to the HTTP session timeout defaults through the <code>SessionTimeoutControl</code> servlet
SESSIONHANDLE, SESSIONREFCOUNT	Not used
SESSION	Traces Documentum session binding and unbinding to HTTP session. SESSION tracing must be enabled for all other tracing flags
REQUEST	Traces Session ID, URL protocol, authorization type, HTTP method, URL scheme, server port, server name, host name, locale, URI, query string, referer
FAILOVER	Traces serialization calls to <code>ReplicatedSession</code> and <code>Container.isFailoverEnabled()</code> . Use this flag to find session attributes that are not serializable. A runtime exception will have a message that a non-serializable attribute has been placed in the session.
FAILOVER_DIAGNOSTICS	Prints the total size of the session in bytes that are serialized at the end of each request. This is a very expensive call and should be used only for debugging purposes.
CLIENTSESSION-STATE	Traces binding and restoring client ID to HTTP session
CLIENTDOCBASE	Traces the repository that the user has navigated to, in login and authentication, multi-repository search results, browser tree navigation, find target in foreign container, Webtop tabbar

Tracing WDK framework operations

The following tracing flags in `com.documentum.web.common.Trace` and `web.formext.Trace` can be used to trace various framework operations:

Flag type	Description
THREADLOCALVARIABLE	Traces binding of variables to threads.
LOCALESERVICE	Traces set locale and create locale hook
ACTIONSERVICE	Traces <code>getActionDef()</code> , <code>queryExecute()</code> , preconditions and results, <code>execute()</code> , return values
CONFIGSERVICE	Traces <code>lookupElement()</code> , inheritance resolution, qualifiers, lookup hooks, <code>IConfigElement()</code>
PREFERENCES	Traces preference setting, caching, and retrieving
ROLESERVICE	Traces role preconditions called by action service
BRANDINGSERVICE, CONFIGTHEMERESOLVER	BRANDINGSERVICE traces the resource folder used for branding; CONFIGTHEMERESOLVER traces theme cache refresh, prints theme list, default theme, CSS for theme, resource folder path
CLIPBOARD	Traces clipboard operations on objects: cut or copy to, paste or paste as link from clipboard, remove, read and write to repository clipboard cache, clipboard ID
MESSAGING	Not used
FOLDER_UTIL	Traces cache updates, hits, and refreshes
PERSISTENTOBJECT-CACHE	Traces caching and retrieval of persistent objects such as DFC objects

Tracing controls and validation

The following flags in `com.documentum.web.form.Trace` can be used after you turn on session tracing:

Flag type	Description
CONTROL	Traces the <code>TreeTag</code> <code>renderEnd</code> event
CONTROLTAG	Traces image folder or file path resolution error, control creation, <code>doStartTag</code> and <code>doEndTag</code> events

Flag type	Description
DATABOUND	Traces query object built, resultset and resultset data handler events, query data handler events, and data provider events
VALIDATOR	Traces the following validators: QuoteValidator, CompareValidator, RangeValidator, InputMaskValidator, DateValidator, and RegExpValidator.
DOCBASEATTR	Traces notifications of start and finish to DocbaseAttributeRequestListener, form empty notification, calls to listener postServerEventControls(), calls to updateAttributeAndDependentList(), traces adding and removing entries to attribute list (can be used to trace duplicate names), and traces the attribute dependency list.
DOCBASEATTRLIST	Dumps the DOM of attributes that are obtained from the data dictionary.
DOCBASESCOPECONFIGSERVICE	Traces operations that read scope and attribute categories from the data dictionary

Tracing JSP processing

Flag type	Description
Forms and form tags	FORM traces URLs for return, nest, and redirect, validation, browser history setting, form binding and unbinding to HTTP session; FORMTAG traces doStartTag and doEndTag events and WebformTag includes. FORMINCLUDETAG is not used.
Page navigation	Several tracing flags give more granularity to form processor tracing: JUMPOPERATION traces onExit event, jumped-to form construction and onInit event, update, change events, and validate; HISTORYRELEASEDOPERATION (not used); NESTOPERATION traces nested form creation and onInit event; PAGEJUMPOPERATION traces update, change, and validate events; RECALLOPERATION traces update, change, validate, and action events; RETURNOPERATION traces onExit; REDIRECTOPERATION (not used); event; TIMEOUTOPERATION (not used); INCLUDEOPERATION (not used);

Flag type	Description
FORMHISTORY	Traces form init and exit, snapshot number, removal, binding, release, recovery
Page processing	FORMPROCESSOR traces form open, URL request, and calls to jump, nest, page jump, recall, redirect, return and jump, and return operations; FORMPROCESSORACTIONS traces form forwarding, rendering, refresh, and form processor hook creation
OPTIMIZE	Times the processing of a form from doStartTag to end of doEndTag.
Form info	FORMINFOCOMMENTS renders form (form class, NLS resource, and URL) or component (component name, config file, vcontext, page, form class, NLS resource, and URL) info into HTML comments; FORMREQUEST traces session state for the form; FORMRESPONSE (not used).
FRAGMENTBUNDLE-SERVICE	Traces bundle cache refresh, storing in cache, bundle not found, base bundle used, fragment tag rendered

Tracing components and applications

Flag type	Description
Components	COMPONENT traces component launching, role precondition check, nesting, form class and NLS bundle, and dispatching context, start page.
CONTAINMENT	Displays the containing form; COMBOCONTAINER traces visitor callbacks, getValue() and setValue() failure
APPCONTEXT	Traces the web application root context and calls to setFolderPath and setFolderId in the AppSessionContext class.
VDMTREEGRID	Traces resynch, create root node, add a node ID to VDM tree grid

Tracing virtual links

The following flags in `com.documentum.web.virtuallink.Trace` can be used to trace virtual linking:

Flag type	Description
VIRTUALLINK	Traces virtual link servlet request, error in repository path, stack trace, authentication request, redirect, invalid 404 error, servlet exception, could not retrieve object from repository, non-sysobject.
VIRTUALLINKCONTENT	Traces Documentum object ID, format error, extraction and mim-type details, file stream initialization, stream written, end of file.
VIRTU-ALLINKMATCHING	Traces virtual link path and path exception, partial match path.
VIRTUALLINKREQUEST	Traces request string and writes details to the log.

Tracing servlets

The following flags in `com.documentum.web.servlet.Trace` can be used to trace servlet operations:

RESPONSE_COMPRESSION	Reports processing times and compression ratio for zipped compression, which is enabled in <code>web.xml</code>
RESPONSE_HEADER_CONTROL	Logs the response headers and session ID
FORMAT_RESOLVER	Traces operations of the file format icon resolver

Tracing asynchronous operations

The following tracing flags in `com.documentum.job.async.Trace` and `com.documentum.job.Trace` can be used to trace asynchronous jobs:

ASYNC_MGR	Traces job operations by the async manager including creating and getting jobs, job events, notification, queueing, timing, execution, and job count
JOB_EXEC_SERVICE	Not used
JOB	Traces job construction and status report

JOB_STATUS_REPORT	Traces job status report requests and messages
JOB_STATUS_LISTENER	Traces calls to the job status listener including job completion percentages, job termination, completion of all jobs

Tracing content transfer

The following flags in `com.documentum.web.form.contentxfer.Trace` and `web.util.Trace`:

Flag type	Description
ZIPARCHIVE	Traces the zip archive that is created to transfer content including stream open, close, file name, and folder name.
GETCONTENT	Traces servlet initiation and completion, object retrieval, format test, extraction, streaming, and writing to local host.

The following tracing flags in `com.documentum.web.contentxfer.Trace` can be used to trace UCF operations:

UCF_MANAGER	Traces UCF server session initialized, new server session ID, server session acquire and release, session disconnect, communication manager registration, and UCF requirement by component (all components with <code><ucfrequired/></code> in definition)
HTTP_MANAGER	Traces add outgoing content file, send outgoing file, remove outgoing file, and set client download event
WDK_API_TRACE	Traces file path for HTTP multi-part file upload

The following tracing flags in `com.documentum.web.common.Trace` and `com.documentum.web.contentxfer` can be used to trace ACF content transfer operations.

CLIENTNETWORKLOCATION	Traces the network location used by the client: get, store, clear preference, get applicable and available network locations
ACS	Traces the reading of the ACS preference (use ACS for HTTP transfer)
CONTENTTRANSFER	Client-side trace flag that provides ACS info for HTTP-based transfer if app.xml is configured to use ACS for HTTP transfer

Adding custom tracing flags

You can add tracing flags specific to your application, and then use the flags as outlined above.

Example 10-1. Adding tracing flags to your application

The following example adds two custom tracing flags to the application.

1. Create a custom trace class that extends one of the WDK trace classes. For example:
2. Add your tracing flags to the custom tracing class as member variables. For example:
3. Add your tracing flags to `com.documentum.debug.TraceProp.properties` in `/WEB-INF/classes/com/documentum/debug`.
4. Use your flag in the application.

```
public class MyTrace extends com.documentum.web.common.Trace
{
    public static boolean MYTIMEOUTCONTROL;

    String timeoutValue = request.getParameter(TIMEOUT_PARAM);
    if (Trace.MYTIMEOUTCONTROL)
    {
        Trace.println("TIMEOUT_PARAM value read: " + timeoutValue);
    }
}
```

Client-side tracing

You can insert client-side tracing in your JSP pages. The client-side tracing JavaScript file `trace.js` is provided in the WDK include directory. To change the JavaScript file that handles client-side tracing output, specify your custom JavaScript file in the `WebformScripts.properties` resource file. The trace JavaScript file, and all other JavaScript

parameters in `WebformScripts.properties`, are rendered into form HTML output by the `WebformTag` class.

To output client-side tracing messages to a popup browser window, call the `Trace_println` function, passing in the message as the sole parameter. For example, within the `<html>` tags of a JSP page:

```
<script language="JavaScript">Trace_println
  ("hello, World");
</script>
```

You can also turn on server-side tracing in a JSP page during development. The trace output will be written to standard output (stdout) and, for some application servers, the output is displayed in the console.

Logging

The open source Apache logging library `log4j` is installed by the DFC installer. This package allows you to enable logging at runtime without modifying the application library or incurring a significant performance impact. The Apache `log4j` library is used by the DFC logger class `DfLogger`. Each `log4j` Logger class method such as `debug()` and `warn()` is wrapped by a `DfLogger` method. The WDK Trace class uses `DfLogger` to write the log file for all enabled traces.

The log file location is specified in a `log4j.properties` file, which is installed by the WDK and client applications installers to `DOCUMENTUM_HOME/config`. By default, the log file name (not path) is specified in `log4j.properties` as the value of the key `log4j.appender.file.File`, for example:

```
log4j.appender.file.File=C:/Documentum/logs/wdk.log
```

You can configure the log file to create a new file each time the log file reaches maximum size by setting the value of `log4j.appender.file.MaxBackupIndex=1` to the maximum number of log files you want, for example, 10 or 20.

To turn on logging of all warnings, change the `rootCategory` property as follows:

```
log4j.rootCategory=WARN, stdout, file
```

To add `DfLogger` entries to the log file, import `com.documentum.fc.common.*` in your class and add log statements similar to the following for tasks that you wish to log. The log statement will be written to `trace.log`:

```
DfLogger.warn ("tracing", "message here", null, null);
```

You can also enable console logging in the `log4j.properties` file. Some application servers automatically route console messages to a log, so this setting may not be necessary for your application server. For information on console logging, consult your application

server documentation. For more information on configuring log4j, refer to the [Apache Web site](#).

You can enable logging of DFC tracing with the log4j package for DFC classes in the file `dfcfull.properties`, which is installed to `DOCUMENTUM_HOME/config`. By default, DFC logging is suppressed. You can turn on tracing of method calls, parameters, return values, trace formatting, tracing depth, and DMCL. You can replace this with the full path and file name. To enable logging of DFC trace calls, change the following line:

```
dfc.tracing.enabled=false
```

to the following value:

```
dfc.tracing.enabled=true
```

The DFC trace file `dfctrace.log` is written to the J2EE server directory that contains the app server executable. For example, in Tomcat the trace file will be located in `C:\Program Files\Apache Tomcat 4.0\bin`.

Performance

There are several application guidelines that can significantly improve performance of your Web application:

- [Action implementation, page 345](#)
- [Documentum object creation, page 345](#)
- [String management, page 346](#)
- [Paging, page 346](#)
- [J2EE memory allocation, page 346](#)
- [HTTP sessions, page 348](#)
- [Preferences, page 349](#)
- [Browser history, page 349](#)
- [Value assistance, page 349](#)
- [Search query performance, page 350](#)
- [High latency and low bandwidth connections, page 350](#)
- [Qualifiers and performance, page 352](#)
- [Import performance, page 352](#)
- [Load balancing, page 352](#)
- [Modal windows, page 353](#)

You can also follow these simple recommendations for performance:

- [Event handling](#)

Server event handling provides code reuse across the application, state management, and better performance.

- Queries

Set `<showfolderpath>` to false in the search component to speed queries.

- Tracing

Turn off tracing to improve performance. Navigate to the page `/wdk/tracing.jsp` and deselect all tracing flags.

Action implementation

The states of all actions associated with dynamic action controls are evaluated when the `actionmultiselect` control is rendered. A large number of selectable items or associated actions can degrade performance. For example, if there are ten selectable items and a hundred associated actions, one thousand states will be evaluated.

Preconditions are called for each item in a list component or `actionmultiselect` control. The action service checks preconditions for each control, and the control tag class renders JavaScript to dynamically show, disable, or hide the controls based on the state of checkboxes. For 10 multiselect items and 50 dynamic actions, this results in a possible 500 precondition calls before page rendering. Action precondition classes must be optimized to manage performance. The `actionmultiselect` control in particular should not have too many selectable items or associated actions.

To reduce the query time for preconditions, you may be able to use a `dm_sysobject` with a custom `a_content_type` attribute instead of a custom object type for type-specific actions.

Another strategy to improve action precondition performance is to cache custom attributes that are used by the precondition by means of a custom attribute data handler. Refer to [Using custom attribute data handlers, page 375](#) for details.

Documentum object creation

Whenever possible, do not call `IDfSession.getObject()`, which performs a fetch of the object. Most attribute arguments can be retrieved without a call to `getObject()`, because they are cached by the initial query on the page rather than from a `getObject()` call. For example, if the page has a databound control to `r_lock_owner`, that attribute value is cached. Your component can check for the existence of the argument value and query only if the argument was not passed.

Queries inside an action class `queryExecute()` method can seriously degrade performance.

String management

The following coding practices can enhance the performance of your application:

- Replace string concatenation using "+" with string buffers, and initialize the string buffer to an appropriate size.
- Strip white space and comments from JSP pages to reduce their size. WDK provides a utility to strip white space and comments: `CommentStripper`, in `WEB-INF/classes/com/documentum/web/tools`. Refer to [Comment stripper](#), page 353 for information on using this tool.

Paging

Controls can retrieve any number of rows from a data provider unless you limit the cache size or apply paging to the datagrid.

Cache size for the number of objects returned by a query is configurable in `Databound.properties`, in `/WEB-INF/classes/com/documentum/web/form/control`. Paging is configured on a JSP page that contains a datagrid. You can limit the choices for page sizes by setting the `pagesizevalues` of the `datapagesize` JSP tag.

The `paged` attribute on the datagrid control provides links that enable the user to jump between pages of data within the enclosing data container. You should page your data for better performance and display. If you set the `paged` attribute to `true`, the data provider or data container will render the appropriate links only if the provider has returned multiple pages of data from the query.

J2EE memory allocation

If the memory allocated to the J2EE server Java virtual machine (VM) is not correctly set, the VM will spend a lot of time destroying Java objects, garbage collecting, and creating new objects. To change the memory allocation, use a setting similar to the following in the Java arguments in the J2EE server start script that was installed by the WDK application installer:

```
-Xms256m -Xmx256m
```

The command elements are described below:

```
1-Xms256m 2-Xmx256m 3-verbose:gc
```

- 1** Starting memory heap size, in MB. In general, increased heap size increases performance up until the point at which the heap begins swapping to disk.

- 2 Maximum Heap size. For a single VM, Sun recommends that you set maximum heap size to 25% of total physical memory on the server host to avoid disk swapping. Increased heap size will increase the intervals between garbage collection (GC), which thus increases the pause time for GC.
- 3 Turns on output of garbage collection trace to standard output (refer to below)

For more information on these settings, refer to your application server documentation.

You can monitor memory usage by the Java process in the Windows task manager to determine whether your memory allocations are optimum.

You can monitor Java garbage collection by setting the command `-verbose:gc` in the J2EE server start script. Increased Java memory settings will increase the amount of time before a major garbage collection takes and will also increase the amount of time that garbage collection takes to execute. Garbage collection is the greatest bottleneck in the application, and all application work pauses during garbage collection.

Garbage collection tracing has the following syntax:

```
[GC 776527K->544591K(1040384K), 0.4283872 secs]
```

The trace can be interpreted as follows:

```
1[GC 2776527K->3544591K(
  41040384K), 50.4283872 secs]
```

- 1 GC indicates minor garbage collection event, Full GC indicates full garbage collection
- 2 Amount of total allocated memory at start of minor collection
- 3 Amount of total allocated memory at end of minor collection
- 4 Amount of total memory on host
- 5 Time in seconds to run garbage collection

Allocated memory as shown in consecutive GC traces continues to grow until full garbage collection occurs. Full garbage collection takes much longer than minor garbage collection, often on the order of 10 times as long.

The following table describes some memory troubleshooting inferences that can be drawn from garbage collection.

Symptom	Reason
Frequent full GC, starting point higher after each full GC, decreasing number of GC between full GC	Total memory too small, or memory leak
Garbage collections take too long (GC 1 sec, full GC 5 sec), server cannot create new threads	Too much memory allocated to JVM

J2EE servers also have configurable settings for thread management which can significantly affect performance. The symptom of insufficient threads is that, as the number of users increases, performance degrades without increased CPU usage. Some users will get socket errors. In Tomcat, the log catalina.log shows that all threads up to maxProcessors have been started, and new requests are rejected with "Connected Reset By Peer." In WebLogic, the execute queue shows waiting threads (0 idle threads, with queue length growing).

The symptom of too many threads is excessive context switching between live threads and degraded response time.

To change the number of worker threads in WebLogic, change the value of Threads Maximum in the Configuration screen for the WDK application domain. Alternatively, you can increase the ThreadCount setting in the WebLogic web.xml file:

```
<Server ListenPort="7001" Name="dctm" NativeIOEnabled="true"
  ServerVersion="7.0.0.0">
  <COM Name="dctm"></COM>
  <ExecuteQueue Name="default" ThreadCount="35"></ExecuteQueue>
</Server>
```

In Tomcat, you can change the maximum number of processors (maxProcessors) /conf/config.xml:

```
<!-- Define a non-SSL HTTP/1.1 Connector on port 8080 -->
<Connector className="org.apache.catalina.connector.http.HttpConnector"
  port="8080" minProcessors="50" maxProcessors="7.0.0.0">
</Connector>
```

HTTP sessions

You can set the maximum number of HTTP sessions for your application in the /wdk/app.xml element <application>.<session_config>.<max_sessions>. When the maximum number of sessions is reached, subsequent requests return a serverBusy JSP page. A value of -1 indicates that there is no limit on the number of sessions.

You can also override the normal J2EE session timeout when the top browser frame is unloaded, such as when the user navigates to another Web site. Instead of the usual 60 minute HTTP timeout, the timeout setting <client_shutdown_session_timeout> is set to

60 seconds when the main (top) window has been closed. Refer to [Number of user sessions, page 112](#) for details.

Preferences

Preferences are stored as cookies. Since cookies are passed back and forth with every request and response, there is a small increase in network traffic. If you need to minimize network traffic or service users on low-bandwidth connections, you may wish to not use preferences.

Browser history

The number of history pages maintained on the server for each window or frame is set by the requestHistorySize flag in the file FormProcessorProp.properties, which is located in /WEB-INF/classes/com/documentum/web/form. The default value is 10. If the value is empty or zero, then history is maintained indefinitely. This setting could significantly affect performance.

Memory that is allocated to maintaining browser history is managed more efficiently on the J2EE server if you generate framesets and frames using the <dmf:frameset> and <dmf:frame> tags. Refer to [Managing frames, page 120](#) for more information.

Value assistance

Performance is affected by the number of value assistance queries to be displayed in the properties component and in other components that display a set of properties. Several interventions will enhance this performance:

- For each value assistance query, turn on the option to allow caching in Documentum Application Builder
- Set the cache_queries option to false (F) in the dcml.ini on the WDK application server host. This will ensure that queries are not persistently cached for a long time, only for the life of the DFC user session.
- Index the associated attributes in Content Server.

Search query performance

In advanced search, you can add a checkbox for case-sensitive search. Set the `casevisible` attribute on the search controls to `true`. Set the default match case as the value of the element `<defaultmatchcase>` (`true` | `false`) in `/wdk/advsearchex.xml`. Case-sensitive queries perform faster.

Note: The 5.3 FAST indexer is not case-sensitive, so the case-sensitive checkboxes are applicable only to non-indexed 5.3 Content Server or 5.2.5 Content Server.

Set `<displayresultspath>` to `false` in the 5.3 search component definition to speed all queries. (Does not query for folder path of each object.)

High latency and low bandwidth connections

Use streamline views (`drilldown`) in applications for high latency connections. The classic (`objectlist`) view is slower.

Two filters are available to improve performance in high latency or low bandwidth networks. The filters are defined as servlet filters in `/WEB-INF/web.xml`. They are turned on by default. The filters are as follows:

- Response compression filter (`CompressionFilter`)

Compresses text responses by mapping requests for `*.jsp`, `*.css`, `*.js`, `*.htm`, `*.html`, and the component dispatcher servlet. If the request `accept-encoding` header indicates that the browser accepts compression, the filter swaps the output stream for a compressed stream in either `gzip` or `deflate` compression formats, depending on which format is accepted by the browser as indicated by the `Accept-Encoding` request header.

The configurable value for this filter, `init-param compressThreshold`, is a size in KB or MB that sets the threshold file size at which output will be compressed. Compression does not decrease the size of the stream for small inputs. Additional, high-bandwidth networks may show improvement for only very large files (hundreds of KB). A value of `3kb` indicates that files 3 KB or larger will be compressed.

Additionally there are `init-params` for turning on compression filter debugging and excluding specific JSP pages from compression filtering.

Limitations:

- Not compatible with all application servers, such as WebSphere 5 or WebLogic 7. Can be enabled on WebSphere by enabling compression on the integrated Apache Web server.

- Browsers Safari 1.0 and IE on Macintosh do not support compression. IE through a proxy does not accept compressed responses. A workaround for this is to set up a transparent proxy that the browser is not aware of.
- There is an unknown CPU cost for the compression.
- Cache control (ClientCacheControl)

Limits the number of requests by telling the client browser and any intermediary caches such as caching proxies to cache static elements such as *.gif, *.js, and *.css files, by adding a Cache-Control response header. After the browser has received a response with this header, it will not re-get the content until the maximum age or until the content is cleared manually from the browser cache.

The configurable value for this filter, `init-param Cache-Control`, is the maximum age in seconds of the static content before revalidation, for example, `max-age=86400` (one day).

Add URL patterns to specify the file types that will be cached. In the following example, *.gif files are cached for up to two days:

```
<filter>
  <filter-name>ClientCacheControl</filter-name>
  <filter-class>com...ResponseHeaderControlFilter</filter-class>
  <init-param>
    <param-name>Cache-Control</param-name>
    <param-value>max-age=172800</param-value>
  </init-param>
</filter>
</filter>
<filter-mapping>
  <filter-name>ClientCacheControl</filter-name>
  <url-pattern>*.gif</url-pattern>
</filter-mapping>
```

Note: Safari browser and IE browsers on the Mac and 5.5 on Windows do not apply this header. Later versions of IE do not support both the cache-control and compression mechanisms at the same time.

Tracing for these filters can be enabled through the standard tracing mechanism (`TraceProp.properties`) or by adding the `debug <init-param>` element to the application deployment descriptor (`/WEB-INF/web.xml`). For example:

```
<filter>
  <filter-name>CompressionFilter</filter-name>
  <filter-class>com.documentum.web.servlet.CompressionFilter</filter-class>
  <init-param>
    <param-name>compressThreshold</param-name>
    <param-value>3kb</param-value>
  </init-param>
  <init-param>
    <param-name>debug</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
```

Qualifiers and performance

Each qualifier that is defined in the application slows performance the first time a component is called. Navigation components must evaluate qualifiers for each action in the component JSP page. To improve performance, remove from your custom app.xml file the qualifiers that your application does not need (the application qualifier is required). In the following example from an app.xml file in the /custom directory, only the type qualifier is used by a custom application. The app qualifier is required for all applications. No components or actions can be scoped to role in this example, because the role qualifier is not defined for the application.

```
<qualifiers>
  <qualifier>com.documentum.web.formext.config.DocbaseTypeQualifier
</qualifier>
  <qualifier>com.documentum.web.formext.config.AppQualifier
</qualifier>
</qualifiers>
```

Import performance

You can limit the number of files that can be imported by a user during a single import operation. This configuration setting is the <max-import-file-count> element with a default of 1000 in the importcontainer component. Extend this component definition to configure a different maximum value.

Certain environments have forward or reverse proxy Web servers that do not support HTTP 1.1 chunking, which is used by UCF for content transfer. For those environments, you must configure UCF to use alternative chunking, and you can tune the chunk size for the Web server. In general, the default chunk size works best for large file transfers. Smaller chunk sizes may enhance performance for small (less than 1MB) files but degrade performance for large files. Refer to [Configuring UCF support for chunked transfer encoding, page 522](#) for more information.

Load balancing

WDK applications can be load balanced using network load balancers. Session "stickiness" (or affinity) must be used. That is, once a session has been established between a browser and a back-end application server then all subsequent traffic from that browser must be routed to that server by the load balancer for the duration of the session. The affinity can be done by IP address or by session cookie depending on the available settings in the load balancing software.

Because content transfer is disk-intensive, best performance spreads the I/O of the WDK content directory over a striped disk volume.

Modal windows

Modal windows provide a performance enhancement in web applications that use several frames. With a modal window, other frames do not need to refresh after the modal frame closes. Refer to [Using modal windows, page 425](#) for more information.

Finding component information

The componentlist component (virtual_root/component/componentlist) displays all of the components in your application. By clicking on a link to a component name, you will see displayed the following information about the component :

- Name of configuration file
- NLS bundle name
- Component parameters
- Whether the component is a container
- Whether the component is configurable
- Fully qualified component class name
- Component description from the component definition

Comment stripper

Your JSP pages will load faster if you strip out white space and comments. A comment stripper tool, CommentStripper, is provided in /WEB-INF/classes/com/documentum/web/tools. This utility is called by the WAR file tool CreateInstallerWAR, so you do not need to use the comment stripper if you are using CreateInstallerWAR (refer to [WAR packaging tool, page 155](#)).

The parameters to use in starting this tool from the console are as follows:

<code>-args <i>filename</i></code>	Remove comments from a single file
<code>-args *.<i>ext</i></code>	Remove comments from all files with the specified extension
<code>-?</code>	Display help

-l	Remove leading white space
-t	Remove trailing white space
-m	Remove HTML comment blocks <!--...--> and <!--...-->
-j	Remove JSP and JavaScript / * ... * / comments
-r	Recurse directories from current
-oxx	Use specified extension instead of overwriting original file
-v	Verbose output (OFF by default)

Testing components

WDK provides a sample library of JSP pages and server classes that test and display all of the WDK controls. The JSP pages are located in the /wdk/samples directory and its subdirectories.

The file wdk/config/fxtest_component.xml defines simple components that test the controls. The component definitions specify the start JSP page that is located in /wdk/samples.

To use a testing component, specify the test component in a URL similar to the following:

```
http://APP_HOME/component/docbaseicontest
```

Additional test suites are provided through two special components:

- testbed_generic and componenttestbed components
- testbed component

Component that can test any action or component from the WDK library, WDK client application library, or custom library built on WDK. This component also provides a common GUI for test automation.

For more information on using the test components, refer to *Web Development Kit Reference Guide*.

Debugging tips

This topic introduces some debugging tips based on error messages that are displayed in the J2EE console or log. Your application server also has documentation on programming

practices specific to the server. There is more information available on JSP, XML, JavaScript, and Java debugging in references specific to the programming tool.

The following topics include debugging tips:

- [Refreshing configuration and data dictionary, page 355](#)
- [JSP debugging, page 355](#)
- [XML debugging, page 356](#)
- [JavaScript debugging, page 356](#)
- [Java debugging, page 357](#)

Refreshing configuration and data dictionary

Changes to the data dictionary and changes to XML configuration files are not refreshed in the application. You must explicitly navigate to the refresh page `/wdk/refresh.jsp` after logging into the application.

If you expect frequent data dictionary changes to your connected repositories, you may wish to set a script that issues the refresh URL at appropriate intervals.

JSP debugging

If the JSP compiler encounters errors in compiling the JSP page, the stack trace can be helpful in pinpointing the error. In the following example, an attribute value in a Documentum JSP tag was malformed. The attribute on the JSP page was "label" instead of "name". The following error was displayed in the browser and logged in the Tomcat server log:

```
2003-06-23 17:05:00 ApplicationDispatcher[/wdk52bis] Servlet.service()
  for servlet jsp threw exception
  org.apache.jasper.compiler.CompileException:
  /custom/main.jsp(25,0)
  Attribute label invalid according to the specified TLD
```

The nature of the error (malformed attribute called "label") as well as the line number and position in the line (25,0) are provided, making it easy to track down the error.

XML debugging

The stack trace in the browser can give helpful information on the type of error in an XML configuration file. In the following example, a typographical error in an element name, `<acomponent>` instead of `<component>`, resulted in this error message:

```
org.xml.sax.SAXParseException: The element type "acomponent"
  must be terminated by the matching end-tag "".
  at com.documentum.xerces_2_3_0.xerces.parsers.DOMParser.parse(
    Unknown Source)
```

This message indicates that the error comes from the XML parser and that the element `<acomponent>` does not have a matching closing tag. This error is returned at startup, when all configuration files are read into memory.

If the error is due to an unknown element, such as `<acomponent>...</acomponent>`, the error is reported as a Java error when the component is called, because the typographical error effectively removes the component definition. The error is reported similar to the following:

```
java.lang.IllegalStateException: Component Definition
  renditions 'component[id=renditions]' does not exist
  within context REQUEST()SESSION(location=homecabinet_streamline)APP()
  at com.documentum.web.fornext.component.ComponentDef.(Unknown Source)
```

A nonexistent start page is reported in a popup window as follows. The path (which is incorrect) is displayed in the stack trace:

```
Stack Trace:
javax.servlet.ServletException:
  /webcomponent/library/properties/custom/library/
  attributes/attributes_dm_document.jsp
```

JavaScript debugging

You can install debuggers for IE and Netscape in order to debug JavaScript errors in your JSP pages, JavaScript included files, or JavaScript that is generated by custom classes. Without a debugger, a JavaScript error is reported in the IE status bar as "Error on Page."

JavaScript errors in which the script calls a non-existent server method or component are displayed in the stack trace in the same way that Java, JSP, and XML errors are displayed. In the following example, a JavaScript function that calls the main component contains a typographical error:

```
window.location.replace("/" + strVirtualDir + "/component/mainX");
```

The following stack trace is displayed:

```
java.lang.IllegalStateException:
```

```
Component Definition mainX 'component[id=mainX]' does not exist  
within context REQUEST() SESSION() APP()  
at com.documentum.web.formext.component.ComponentDef.(Unknown Source)
```

Java debugging

Java debugging is provided through integrated development environments (IDEs) and compilers. Consult the documentation for your compiler or IDE (if the IDE includes a compiler) for details on debugging your Java code.

Component, Action, and Control Design Guidelines

This chapter presents guidelines and checklists for developing a component.

The following topics describe guidelines and checklists for development:

- [General guidelines, page 359](#)
- [Design checklists, page 362](#)
- [Component checklist, page 367](#)
- [Component unit test checklist, page 376](#)
- [Control checklist detail, page 365](#)
- [Component checklist detail, page 373](#)

General guidelines

The following general guidelines describe conventions that apply to file names and locations, accessibility, and internationalization:

- [File follows naming convention, page 359](#)
- [File follows location convention, page 361](#)
- [Follows accessibility guidelines \(Section 508\), page 362](#)
- [Externalizes and tests strings, page 362](#)

File follows naming convention

The following naming conventions are used in the WDK libraries, as a suggested model for custom libraries. *Italics show variables*, and [] show optional elements:

- Action configuration file

object type_actions.xml For example: *dm_folder_actions.xml*.

Action names must be all lowercase, and where practicable, the first word of any action name should be a verb, the second a noun. For example: *importrendition*

All actions that are pertinent to an object type should be defined in one actions definition file.

- Action behavior file

Actions class names should use mixed case, with the first and each subsequent word capitalized. The naming syntax is *[Verb][Noun]Action.java*. For example: *DeleteQueueItemAction.java*

Precondition class name should use mixed case, with the first and each subsequent word capitalized. The naming syntax is *[Verb][Noun]Precondition.java*. For example: *DeleteDocPrecondition.java*

All action package names must be entirely lowercase and should follow the location of the action definition file. For example: *:com.mycompany.actions*

- Component configuration file

operation_[object type]_component.xml. For example: *delete_notification_component.xml*.

Component names must be all lowercase; and where practicable, the first word of any component name should be a verb, the second a noun. For example: *checkin*

For components that are containers, "container" should be appended to the name. For example: *checkincontainer*

Only one component should be defined in one component definition file.

- Component behavior file

Component class name should resemble the component ID and use mixed case, with the first and each subsequent word capitalized. For example: *CustomSubscription.java*, which matches the *customsubscription* component.

All component package names must be entirely lowercase. For example: *com.mycompany.subscription*

The first word of any event handler method should be 'on'. For example: *onShowOptions()*

- Component unit test file

Test class name should be the same as the behavior class name and be prefixed with "Test". For example: behavior class *NewCabinet*, test class *TestNewCabinet*.

- Component JSP file

Name should be descriptive, contain no spaces, and use mixed case, with each word capitalized except the first letter of the first word. For example: *customSubscription.jsp*

- NLS file

NLS properties file names should use mixed case with the first and each subsequent word capitalized, and have the following name syntax: *ActionNameNlsProp.properties*, or *ComponentNameNlsProp.properties*. For example: *CustomSubscriptionNlsProp.properties*

Follow Java's ResourceBundle naming convention in order to generate localized versions of the properties files. Refer to [Naming properties files, page 140](#) for details.

- Control class

All control package names must be entirely lowercase and end with control, for example, *com.mycompany.control*.

A control class name should be descriptive, contain no spaces, and use mixed case with the first and each subsequent word capitalized, for example, *Image.java*, *Label.java*. The corresponding tag class should start with the control name and end with Tag, for example, *ImageTag.java*, *LabelTag.java*. All tag and attribute names must be entirely lowercase, for example, `<mylib: specialtag width='100'>`.

File follows location convention

The following location conventions are used in WDK libraries, as a suggested model for custom libraries:

- Action configuration file:

Action configuration files reside in an actions folder. For example:
T:\app\custom\config\actions

- Action behavior file

Java class packages will dictate where source codes and compiled classes will reside.

- Component configuration file

Component configuration files reside in a config folder under the library folder.
For example: /custom/config

- Component behavior file

Java class packages will dictate where source codes and compiled classes will reside.

- Component unit test file

Java class packages will dictate where source codes and compiled classes will reside. If test classes require access to protected methods, place them in a separate parallel directory structure with package alignment (recommended for all test assets):

```
src
  com
    mycompany
```

```
    attributes
      Attributes.java
test
  com
  mycompany
  attributes
  TestAttributes.java
```

- JSP pages:

JSP files reside in a component folder. For example: /custom/checkin

All JSP files that are pertinent to a component should reside in the same component folder.

- NLS properties file

NLS properties file reside in a /strings folder in a path similar to the associated component or action class location. For example:T:\app\custom\strings\com\mycompany\checkin. The associated component class is located in /WEB-INF/classes/com/mycompany/checkin.

Follows accessibility guidelines (Section 508)

Follows guidelines on complying with U.S. government Section 508 accessibility requirements. Refer to "Accessibility Service" in *Web Development Kit and Client Applications Development Guide* for WDK guidelines. See also [US government guidelines](#).

Externalizes and tests strings

Components, actions, and control should externalize all strings used in the UI and in error messages to the user or log using properties files. For information on internationalization and localization practices, refer to the documentation on properties files and the locale service in *Web Development Kit Reference Guide*.

Design checklists

Use the following checklists as you develop WDK-based components, actions, and controls:

- [Control checklist, page 363](#)
- [Control checklist detail, page 365](#)
- [Component checklist, page 367](#)

- [Component checklist detail, page 373](#)
- [Component unit test checklist, page 376](#)

Each checklist table has the following columns:

- Checklist item
Describes the design feature
- Impact
Rates the importance of the checklist item for the success of the feature:
 - High: Not following this item will cause lack of reusability, buggy behavior, inability to migrate, or failure of the component to run
 - Medium: Not following this item will impact the success of the feature but will not cause disastrous consequences
 - Low: A standard or design practice that facilitates the reuse of the component but will not affect the success of the component itself
- Significance
Describes the type of impact for this design feature. For example, Reuse indicates that not following this checklist item will impact the ability of this component to be reused.
- Followed Y/N
Use this column to check off whether you have followed the checklist item or not.
- Justification if ignored
Write a justification of why a particular checklist item has not been followed.

Control checklist

The following checklist provides a summary of the guidelines for WDK controls and offers a quick test of whether the guidelines have been followed. Follow the link in the Checklist Item column to jump to more detailed information about a specific guideline.

Note: All items are required unless labeled "Best Practice."

Table 11-1. Control checklist

Item	Impact	Signifi- cance	Followed Y/N	Justification if ignored
Control is named using a constant defined in behavior class	L	Reuse		
Control is defined at design time: Appearance in the UI State Properties Events that it fires	H	Standard		
Creates New Control if Needed. Creating new control if needed, page 365	H	Migration		
Control and tag class follow naming convention. File follows naming convention, page 359	M	Reuse		
The namespace for the control clearly classifies the control into an appropriate functional area/application: Related to specific functional area/application, placed in appropriate control library (not Documentum)	H	Reuse		
Uses base tag class rendering helpers to render HTML tags that conform to WDK protocol	H	Quality		
Implements the tag class release method or calls super.release.	H	Quality		
Control class property getters and setters are type safe. For example, boolean values are passed/returned for boolean properties.	H	Extensibility Usability		

Item	Impact	Significance	Followed Y/N	Justification if ignored
Uses a theme style (skinnable), tested against supported schemes	H	Reuse		
Formats and escapes all values to render well formed HTML using <code>ControlTag.formatText()</code> or <code>formatAttribute()</code>	H	Security Quality Reuse		
Externalizes and tests strings, page 362 . Externalizes and tests strings, does not cache strings that will be localized.	H	I18N		
Rendering JavaScript: Ensures that page is loaded and initialized. Ensuring that page is loaded and initialized, page 367	H	Quality		
Creates a control pen and tests control attributes	H	Quality		

Control checklist detail

The following guidelines apply to controls:

- [Creating new control if needed, page 365](#)
- [Using base tag rendering helpers, page 366](#)
- [Formatting and escaping rendered HTML, page 366](#)
- [Ensuring that page is loaded and initialized, page 367](#)

Creating new control if needed

Does not use code to generate HTML controls. Instead, uses existing controls or constructs new custom controls based on WDK control framework. This leverates the WDK programming model of handling controls, such as rendering, event handling, state management.

Using base tag rendering helpers

The rendering helpers render HTML tags that conform to WDK protocols:

- `renderEventHREF()` and `renderEventCall()`
- `renderNameAndId()`
- `renderStartToolTip()` and `renderEndToolTip()`



Caution: Don't render HREFs directly. Doing so can cause servlet write connection exceptions in certain application or portal servers such as BEA WebLogic.

Formating and escaping rendered HTML

All rendered HTML values should be formatted and escaped to render well-formed HTML that is not vulnerable to cross-site scripting, a security risk in which JavaScript can be entered as a value. (For information on `app.xml` configuration to detect URL parameters susceptible to cross-site scripting, refer to [<requestvalidation> element, page 71.](#))

Use the `ControlTag` class methods `formatText()`, for a string, or `formatAttribute()`, for an HTML attribute, to generate a safe string. The following example from `BookmarkLinkTag` formats a label:

```
buf.append(formatText(link.getLabel()));
```

The following example from the `DateWithTag` class renders time as a text control:

```
strTime = getString("MSG_TIME");  
buf.append(" value='").append(formatAttribute(strTime)).append("'");
```

The following type of URL (or the scripting portion of it) can be used to test cross-site scripting vulnerability of component parameters or user input controls (line is broken for display purposes):

```
http://hostname:port/webtop/component/tabbar?entryTab=>'<<script>alert('Appscan%20-%20CSS%20attack%20may%20be%20used')</script>&Reload=1107912068508&__dmfFrameId=Streamline_tabbar_0  
  
http://hostname:port/webtop/component/tabbar?entryTab=>%22%27>  
<img%20src%3d%22javascript:alert(%27Appscan%20-%20CSS%20attack%20may%20be%20used%27)%22>&Reload=1107912068508&__dmfFrameId=Streamline_tabbar_0
```

Encoding for a particular label control can be enabled or disabled by calling `setEncodeLabel()` on the control from your component class. For information on global enabling or disabling of attribute encoding, see [Display of escaped HTML strings, page 201.](#)

Ensuring that page is loaded and initialized

JavaScript should not be executed until after the page had been fully initialized. The act of initialization is performed in concert by the client browser and the server. If the JavaScript code uses the `safeCall()` method, then initialization is assured. The JavaScript code can also use the `isWindowInitialised()` method to determine whether initialization is complete. If initialization is not complete, then the control rendering code should fail gracefully. In the following example, a component class generates JavaScript that includes a call to `isWindowInitialised()`:

```
buf.append("<script>");
buf.append("function _showmenu() {if (isWindowInitialised(
    window) == true) {popupMenu(' + MENU_ID + "');} else {setTimeout(
    '_showmenu();',250);} }");
buf.append("</script>");
```

Component checklist

The following checklist provides a summary of the guidelines for WDK components and offers a quick test of whether the guidelines have been followed. Follow the link in the Checklist Item column to jump to more detailed information about a specific guideline.

Note: All items are required unless labeled "Best Practice." Follow the hyperlink for certain checklist items to find more information.

Table 11-2. Component design checklist

Item	Impact	Signifi- cance	Followed Y/N	Justification if ignored
Creates a new XML configuration file for this component (doesn't modify existing file)	H	Migration		
Creates a new JSP file if page modifications are needed	H	Migration		
File follows naming convention. File follows naming convention, page 359	M	Reuse		
File follows location convention. File follows location convention, page 361	H	Reuse		

Item	Impact	Signifi- cance	Followed Y/N	Justification if ignored
Extends rather than copies original component definition	M	Reuse Extensibility		
Describing a component , page 374 Definition describes component using <desc> value, including why component is being extended or created	M	Reuse		
Making a component configurable , page 374 Contains values that can be configured to change behavior rather than requiring customization. Includes comments explaining what is configurable and possible values.	H	Reuse Extensibility		
Required parameters are marked <required>true</required>	H	Reuse		
Has help ID defined. Make help ID = component ID	M	Usability		
Making the component definition backward-compatible , page 374 If component definition is inherited, does not add or remove required <params>. Did not remove structural elements from configuration data.	H	Migration		

Table 11-3. JSP page checklist

Item	Impact	Signifi- cance	Followed Y/N	Justification if ignored
No Java code in JSP: Uses existing or new controls rather than code in JSP	H	Maintain- ability		
JavaScript: Uses safeCall(...) and isWindowInitialised() helpers. Ensuring that page is loaded and initialized, page 367	H	Quality		
File follows naming convention. File follows naming convention, page 359	M	Reuse		
File follows location convention. File follows location convention, page 361	H	Reuse		
Follows section 508 accessibility guidelines. Follows accessibility guidelines (Section 508), page 362	H	Standard Usability		
Uses an HTML element ID instead of Form[0]. Form-indexed elements do not work in portal environment.	H	Reuse		
Externalizes and tests strings, page 362 . Externalizes and tests strings	H	I18N		
Uses WDK control tags, not HTML input controls	M	Quality Portability		
Uses WDK layout tags such as body, header in place of HTML tags	H	Reuse in portal		
Uses a background style such as contentBackground on a body tag. Other styles can bleed through.	H	Reuse		
Uses UTF-8 encoding	H	I18N		

Item	Impact	Signifi- cance	Followed Y/N	Justification if ignored
Uses WDK 5 error directive	H	Reuse		
Uses WDK theme styles. Tests component with all themes defined in app.xml (component is 'skin worthy')	H	Reuse Usability (consis- tency) Configurability		
Layout is consistent with other components in library and is provided by and reviewed by UI group	H	Reuse Usability		
Is compilable: a tag library is defined for the JSP	H	Quality Reuse		

Table 11-4. Behavior class checklist

Item	Impact	Signifi- cance	Followed Y/N	Justification if ignored
If not a container, does not contain container controls such as a title or OK, Cancel, or Help buttons. Container component extends an existing container if possible.	H	Reuse		
Creates a new behavior class file if modifications necessary	H	Migration		
File follows naming convention. File follows naming convention, page 359	M	Reuse		
File follows your component library location conventions	H	Reuse		
Class follows section 508 accessibility guidelines. Follows accessibility guidelines (Section 508), page 362	H	Usability		
Extends a component class rather than duplicates it	H	Migration Extensibility		

Item	Impact	Signifi- cance	Followed Y/N	Justification if ignored
Does not have context-sensitive behavior: Uses WDK filter to configure component behavior in XML definition. Removing context-sensitive behavior from the class, page 375	H	Reuse Extensibil- ity		
Data is not globally cached. Caching data, page 375	H	Reuse		
Designed for subclassing: Applies "template method" pattern, provides hooks for component configuration (class implements a "hook" interface" for complicated application logic)	H	Extensibil- ity		
Contains presentation logic only, and invokes encapsulated business logic (DFC SBOs or TBOs). When this is not possible, encapsulates business logic in the behavior class.	H	Exten- sibility Interoperability		
Uses the asynchronous job execution framework if possible: pre- and post-processing of business logic, business logic runs asynchronously.	H	Exten- sibility Performance		
Binds datagrids with a session on every render method (session is only valid for one request)	H	Quality		
Uses getControl() instead of caching/holding control instances as member variables	L	Quality Extensibility		
Calls super.xxx on all component lifecycle events (onInit, onRender...)	H	Quality		

Item	Impact	Significance	Followed Y/N	Justification if ignored
Uses request scope instead of component scope as parameter for <code>component.getSession</code> rather than <code>DFC sessionmanager.getSession</code>	H	Performance (scalability)		
Does not do the following: Remove non-private methods, constants, member variables Change signature Change behavior of methods or interface Break previous unit tests	H	Migration		
Uses custom attribute data handlers. Using custom attribute data handlers, page 375	H	Performance		
Uses getters and setters for accessing and setting properties.	H	Extensibility Reuse		
Instances of action and precondition classes do not have state associated with them.	H	Quality		
Externalizes and tests strings, page 362 . Externalizes and tests strings, does not cache strings that will be localized.	H	I18N		
Follows DFC guidelines for DFC objects in WDK classes: no API calls, creates all objects using <code>IDfClientX</code> . Following DFC guidelines, page 376	H	Migration Reuse		

Table 11-5. Internationalization checklist

Checklist Item	Impact	Signifi- cance	Followed Y/N	Justification if ignored
Creates a new NLS Properties file for any new strings	H	I18N		
NLS properties file follows naming convention. File follows naming convention, page 359	H	I18N		
NLS properties file follows location convention. File follows location convention, page 361	H	I18N		
Uses NLSINCLUDE rather than duplicating strings	M	Localiza- tion cost Extensibility		
Uses "MSG_xxx" as naming convention for resource strings	L	Standard		
Escapes non-Latin characters	H	Quality		
Does not use sentence/text fragments as values for resource strings	H	Localiza- tion cost Quality		
Uses Preferences debug settings to run NLS checks to make sure strings in XML definition, JSP page, and component class are externalized.	M			

Component checklist detail

The guidelines in this section provide clarification or examples for some of the items in the checklists:

- [Describing a component , page 374](#)
- [Making a component configurable, page 374](#)
- [Making the component definition backward-compatible, page 374](#)
- [Removing context-sensitive behavior from the class, page 375](#)

- [Caching data, page 375](#)
- [Using custom attribute data handlers, page 375](#)
- [Following DFC guidelines, page 376](#)

Describing a component

The `<desc>` tag in the XML configuration file should contain a summary of the component. It should NOT be a restatement of the component name. While the description does not affect the functionality of the component itself, it is extremely important for reus. It should provide enough information to help a developer decide whether the component is a candidate for reuse when scanning a list of descriptions such as those generated by the `componentlist` component.

The following are bad and good examples of the description for a component called `checkin_component.xml`:

Poor: `<desc>Component to check in objects. </desc>`

Better: `<desc>You can use the checkin component to allow users to do the following: 1. Check in previously checked-out components. 2. Set various properties, such as version numbering, locking, and format.</desc>`

Making a component configurable

Contains values that can be configured to change behavior of components rather than requiring customization. For example: Allows configuration of whether major and/or minor versioning is allowed for `checkin`, which columns are visible, and more.

Comments should be included to explain what is configurable and the associated possible values.

Making the component definition backward-compatible

Does not add or remove required `<params>`. Adding or removing required params of extended components with new names will affect backward compatibility.

Adding or removing optional params will not affect backward compatibility.

Removing context-sensitive behavior from the class

A behavior class should use a WDK filter in the component definition to configuration context-sensitive behavior. For example, the following filter applies a context-sensitive scope to the configuration option enableShowAll:

```
<filter role='administrator'>
  <enableshowall>true</enableshowall>
</filter>
```

In the behavior class, the filter is applied as follows:

```
Boolean bEnableShowAll = false;
Boolean enableOption = lookupBoolean("enableshowall");
if (enableOption != null && enableOption.booleanValue == true)
{
    bEnableShowAll = true;
}
```

Note that the boolean variable bEnableShowAll will be set to true only if the user has the role of administrator.

Caching data

A component class should use component arguments and return values to pass data to another component. Avoid using HTTP sessions or static variables. If you need to pass data between components, perform the following steps:

- The calling component should generate a unique variable name within the current user session and store the object with SessionState under this unique name. The caller component should remove the object from the "SessionState" upon return from the calling component
- Add a required parameter to the calling component that specifies the variable name that holds the object in the SessionState

Using custom attribute data handlers

The interface ICustomAttributeDataHandler facilitates more efficient queries for custom data for most, if not all, pages in WDK. Refer to [Adding custom attributes to a datagrid, page 407](#). for details.

Following DFC guidelines

Do not use `apiGet()` or `apiSet()` calls. API calls are deprecated in Content Server 5.3. Instead, use the appropriate DFC method. For example, the following call uses `apiGet`:

```
strId = getDfSession().apiGet("get", strObjId + ",i_folder_id, 0");
```

This call should be changed to the following:

```
IDfSysObject sysobj = (IDfSysObject) session.getObject(objectId);
strId = sysobj.getFolderId(0);
```

Do not create DFC objects such as `DfId`, `DfQuery`, or `DfList` directly. Instead, create the object using `IDfClientX`. For example, to create an ID, use the following:

```
IDfClientX clientX = new DfClientX();
IDfId myId = clientX.getId(string);
```

Do not use `DfTypedObject` or classes that are derived from it. Instead, use `IDfTypedObject` APIs. In the following example, the session manager returns the interface which is then used to access other objects:

```
IDfSessionManager mgr = SessionManagerHttpBinding.
    getSessionManager();
IDfSession dfSession = null;
...
dfSession = mgr.getSession(strDocbase);
IDfTypedObject config = dfSession.getDocbaseConfig();
boolean bInstalled = config.getBoolean(MEDIASERVER_INSTALLED);
```

Component unit test checklist

This checklist assumes the unit test harness is JUnit with the Cactus extension (used for testing server-side Java code) and that a test infrastructure has been established for creating Documentum sessions, creating Documentum objects, data driving tests, etc.

Table 11-6. Unit test checklist

Item	Impact	Signifi- cance	Followed Y/N	Justification if ignored
Creates a new test behavior class file	H	Standard		
File follows naming convention. File follows naming convention, page 359	L	Quality		

Item	Impact	Significance	Followed Y/N	Justification if ignored
File follows location convention. File follows location convention, page 361	L	Quality		
For each public method in the behavior class, a corresponding test method exists in the test class.	H	Quality		
(Optional) For each protected method in the behavior class, a corresponding test method exists in the test class and is prefixed with "test".	L	Quality		
Where feasible, a test should exist to exercise all branches of code in behavior class methods.	M	Quality		
Valid parameter values should be passed to behavior class methods and no errors should occur.	H	Quality		
Invalid parameter values should be passed to behavior class methods and exceptions should be gracefully handled.	H	Quality		
All test classes compile with no errors or warnings.	H	Quality		
All JUnit tests should pass and there should be no errors in the app server console window.	H	Quality		

Table 11-7. Component functional testing checklist

Item	Impact	Significance	Followed Y/N	Justification if ignored
The component can be launched standalone in a browser (unless it extends combocontainer, which must be launched by the LaunchComponent action class).	H	Quality		
The component can be launched from within the parent application in a browser (e.g. from within Webtop).	H	Quality		
When launched with invalid parameters, the component gracefully handles the exception.	H	Quality		
The core behavior of the component works correctly and is in accordance with the functional specification.	H	Quality		
Core behavior of the component works correctly against the current version of the Content Server as well as the last version of the Content Server.	H	Quality		

Customizing Controls

Controls are represented by a class that extends `com.documentum.web.form.Control`. Each control has a corresponding tag represented by a class that extends `com.documentum.web.form.ControlTag`.

Controls are found in two WDK packages: basic HTML controls in `com.documentum.web.form.control` and repository-enabled controls in `com.documentum.web.formext.control`. The `webcomponent` layer does not contain any controls. Webtop and Web Publisher add controls in the `com.documentum.webtop.control` and `com.documentum.wp.control` packages, respectively.

For information on the configurable attributes for individual controls, refer to *Web Development Kit Reference Guide*.

The following topics describe control APIs and their use in custom applications:

- [Control classes, page 380](#)
- [Using controls programmatically, page 382](#)
- [Programming databound controls, page 389](#)
- [Generating UI, page 409](#)
- [Generating a link in a control, page 410](#)
- [Making a control accessible to JavaScript, page 411](#)
- [Displaying folder paths and breadcrumbs, page 412](#)
- [Implementing multiple selection, page 415](#)
- [Managing control events, page 416](#)
- [Validating a control value, page 428](#)
- [Validating a repository object, page 429](#)
- [Adding a control listener, page 430](#)
- [Control lifecycle events, page 428](#)
- [How controls and tags work together, page 432](#)
- [Control arguments, page 433](#)

Control classes

The Control class implements member variables, server-side getter and setter methods, and event handlers for a control. The control tag class initializes the control, provides setter and getter methods on the control members, and renders HTML and JavaScript output for the control. Use the control class methods to change the values of a control's properties. Use the tag class methods to programmatically change the UI.

[Control arguments, page 433](#) maintain the control constants and pass event arguments.

The Control class in the `documentum.web.form` package is an abstract class. Each control object is contained by a container (parent control). The control class exposes type-safe methods to access contained controls and the parent container.

The Control class performs the following functions:

- Exposes methods to get and set control properties
- Implements `getEventNames()` to list the events raised by the control
- Implements `updateStateFromRequest()` and `hasChanged()` to support input data from the client

Control properties are modeled as Control class member variables. Control instances on the server are bound to successive requests, so that member variable values are preserved across requests.

Properties are named and manipulated by public access methods. Custom controls must provide methods that set and get the properties of the control.

The following table describes properties that are common to all controls:

Table 12-1. Control class properties

Property	Description
<code>name</code>	String that identifies the control so that it can be manipulated by server-side event handlers
<code>nlsid</code>	National Language Support (NLS) ID, which is used by <code>Form.getString()</code> to look up a localized string. The string is displayed as a UI element.
<code>datafield</code>	Name of a data column in a recordset that contains data. The control that obtains data from a datafield must be embedded inside a dataprovider control such as <code>datagrid</code> .
<code>elementName</code>	Name of a control when it is rendered as an HTML element

Property	Description
ID	Identifier that is generated by the framework to identify the control
enabled	Boolean attribute that specifies whether the control is enabled
visible	Boolean attribute that specifies whether a control is visible in the UI

The WDK tag classes extend two JSP tag classes: TagSupport and BodyTagSupport. The table below describes the WDK base tag classes and their uses.

Table 12-2. Base tag classes

Class	Use
ControlTag	Extends javax.servlet.jsp.tagext.TagSupport. Binds and renders the control class instance to the UI (JSP page) and renders the control's layout into HTML and JavaScript. The ControlTag class instance lasts only within the lifetime of the HTTP request. Extend this class when your control does not accept user input and does not need to process tags contained within it. Overrides renderStart() and renderEnd() to render HTML. For example: Button, Image, Link, Label.
BodyControlTag	Extends javax.servlet.jsp.tagext.BodyTagSupport. Extend this class when your control does not accept user input but does process tags inside it. Overrides javax.servlet.jsp.tagext.BodyTagSupport methods for HTML rendition. For example: DatagridRow, Panel, NoDataRow.
StringInputControlTag	Extends ControlTag. Extend this class when your control accepts a string from user input but doesn't need access to tags contained within the control. The <i>value</i> attribute accepts user input. The corresponding control class must extend StringInputControl. Overrides renderStart() and renderEnd() to render HTML. For example: Text, Password, Hidden.
BooleanInputControlTag	Extends ControlTag. Extend this class when your control accepts a Boolean input from the user but doesn't need access to tags contained within the control. The <i>value</i> attribute accepts user input. Overrides renderStart() and renderEnd() to render HTML. The corresponding control

Class	Use
	class must extend BooleanInputControl. For example controls: Checkbox, Radio.

The ControlTag class performs the following functions:

- Exposes setter methods for tag attributes
- Implements getControlClass() and setControlProperties(). These methods are called when the control is first rendered, so that the framework can create the control.
- Implements renderStart() and renderEnd() to generate the HTML rendition of the control

Custom controls must implement the release() method in order to maintain the state of a control. Some J2EE servers use the same instance of a tag class for all instances of the tag on a JSP page.

The control rendition code is implemented in the tag classes that extend ControlTag. The rendition methods renderStart() and renderEnd() are defined by the super class.

The tag library specifies whether a control can process body content with the bodycontent attribute. If the value for the bodycontent attribute in the tag library entry is "empty", the control will not process body content. If the value is "jsp", JSP content between the start and end tag will be processed. In the following example, the argument tag bodycontent value is "empty", so the tag contains no content:

```
<dmfx:argument name='objectId' contextvalue='objectId' />
```

A tag that contains JSP content has a start and end tag. For example:

```
<dmf:nodataRow>
  <td><dmf:label nlsid='MSG_NO_DOCUMENTS' />
</dmf:nodataRow>
```

Tag attributes correspond to control class properties. The control tag caches the attribute values and then sets the control properties by calling Control.setControlProperties().

You can set or get control values using methods on the control tag class, but do not set default values in your tag class or they will override the values that you try to set programmatically. Instead, set default values using setControlProperties() or using the tag attributes in the JSP page. The first time the control is rendered, the framework initializes member variable values by calling setControlProperties(). If you set a value for a control tag in a JSP page, the value takes precedence over a value set programmatically.

Using controls programmatically

You can use controls programmatically in the following ways:

- [Creating controls, page 383](#)

- [Naming and getting controls, page 384](#)
- [Setting control values, page 386](#)
- [Getting datagrid controls, page 388](#)
- [Passing arguments to action-enabled controls, page 389](#)

Creating controls

There are three ways to create a control:

[Creating a control in a non-component class, page 383](#)

[Creating a control in a component class, page 383](#)

[Creating a control in a tag class, page 384](#)

Example 12-1. Creating a control in a non-component class

You can create a control using the *new* operator. Controls require no constructor arguments.

The following example from Webtop creates a tab control, sets its name, label, and parent form, and adds it to a tab bar control:

```
Tab tab = new Tab();
tab.setForm(this);
tab.setValue(strComponentLabel);
tab.setName(strComponentId, 0);
Tabbar tabs = (Tabbar)getControl(TABBAR_CONTROL_NAME);
tabs.addTab(tab);
```

Example 12-2. Creating a control in a component class

You can create a control using `Form.getControl()`, passing in the name of the control and the control class name. If the requested control does not exist, it is created. If you use the single argument for `getControl()`, without the control class argument, the control must already exist.

The following example from the Webtop About class gets the Webtop version control, looks up the version value from an NLS resource file, and sets the version value:

```
ResourceBundle bundle = ResourceBundle.getBundle(WEBTOP_BUILD_PROPERTIES);
if (bundle != null)
{
    String strBuild = bundle.getString("Build");
    if (strBuild != null && strBuild.length() != 0)
    {
        Label lblBuild = (Label)getControl(CONTROL_BUILD, Label.class);
        lblBuild.setLabel(strBuild);
    }
}
```

Example 12-3. Creating a control in a tag class

The control tag class implementation creates a control when the control is first rendered. The tag class extends `ControlTag`, which implements `getControl()` to look up a control by name. If the control doesn't exist or doesn't have a name, `getControl()` calls `createControl()`, and a new instance of the control is created for every rendition of the form. The tag class then calls `setControlProperties()`, passing in the new control object.

Your implementation of `setControlProperties()` should set tag attributes as properties of the control object.

In the following example from the Webtop control tag class `DocbaseSelectorTag`, the `renderStart()` method initializes the control and gets the visibility of the control from the tag arguments:

```
DocbaseSelector selector = (DocbaseSelector)getControl();
if (selector.isVisible() == true)
{ ... }
```

Naming and getting controls

A control is rendered into one or more HTML elements. A control can be identified by its name, by a reference to the control object in server memory, by the name of its root HTML element, by the name of a specific HTML element, or by the ID of a specific HTML element.

In a JSP page, set the name attribute on the control tag. The control name must contain only JavaScript symbols A-Z, a-z, 0-9, and underscore. For example:

```
<dmf:text name="my_text" .../>
```

You can also set the control name programmatically by calling `setName(String strName)`.

When a control is named, server code and JSP code can access the control by calling `getControl(String strName)`. Named controls are retained in server memory and bound to each HTTP request, so that the control's state is maintained between HTTP requests. Controls that deliver input, such as a text box, must be named. Controls that are used for display, such as a label, or for raising events, such as a button, should be named if you need to access the control state on the server.



Caution: If you do not name a control, it will not retain state information when the user navigates through browser history. This can cause unexpected errors such as the wrong event handler being called on a control.

Example 12-4. Retrieving a control value

You can retrieve the value of control attributes by using accessor methods for the controls.

The following example gets the values of a checkbox control on a JSP page in the component:


```
Checkbox control = (Checkbox)getControl(
    SEQUENTIAL_CHECKBOX_CONTROL_NAME);
retval = control.getValue();
```

The next example gets the value of a text control:

```
String accessorNameForQuery = ((Text) getControl(
    jumptotextbox)).getValue();
```

Within a Web application, each control has a unique ID and unique name. The name is formed from a root name and an index. Each control maintains an index of contained controls by name and ID. Thus, a component (which is a specialized control) has an index of all of its named controls. The name index contains the names of the child controls. The ID index contains the IDs of all generations of contained controls.

The ID is not used by the WDK framework, but you might need to use it in JavaScript. JavaScript doesn't have access to the form name, so you must access the control by HTML ID. You can set the ID programmatically by calling `setId(String strId)`.

The control ID is set by the framework when you have specified an ID attribute on a control tag. If you do not give the control an ID, the framework generates an ID that is a combination of the form name, control name (or control type, for unnamed controls), and index, for example, `Login_username_0`. If a control is not named in the JSP tag attributes, the form or control class name is used.

Example 12-5. Getting a control by ID in JavaScript

In the following example from `advSearch.jsp`, a text control is given an ID:

```
<dmf:text name='location' id='location' size='70' defaulttonenter='true'
    tooltipnlsid="MSG_LOOK_IN" />
```

The control is retrieved in JavaScript in the same page using the JavaScript function `document.getElementById()`:

```
<script>
    if (document.getElementById("location") != null)
        document.getElementById("location").focus();
</script>
```

Example 12-6. Getting the value of a hidden control

When multiple controls on a JSP page have the same name (for example, controls in a data grid) the controls are automatically assigned an index number. The numbers start with zero and are assigned to the controls in the order that the controls are created. You can pass in an index number when you call `getControl()`.

The following example processes a click on a link in a data grid. The `onClickLink()` event handler gets the index of a hidden control and then gets the object ID of the object::

```
public void onClickLink(Link link, ArgumentList, arg)
{
    //Get the index of the link that was clicked
    int nIndex = link.getIndex();
```

```
//Get a corresponding hidden control with object ID as its value
Hidden hidden = getControl("r_object_id", nIndex);
IDfId id = new DfId(hidden.getValue());
...
}
```

Setting control values

In your component behavior class, you can get a control by name. You can set the control state before the control is rendered. To do this, call `getControl()`, passing the control class to the component `onInit()` class. Then get or set values in the control. using the class `setX` methods.

Note: To get the control in server-side code, the control must be named, either in a JSP tag or programmatically. To set the control name programmatically, call `setName(String strName)`.

In the control tag class, which renders the control into HTML, set your default control properties using `setControlProperties`. Any properties that are set on the JSP page will override these default properties.

Example 12-7. Initializing control properties

The following example from a button tag class initializes the width, height, label, event handler, and default value on a button:

```
protected void setControlProperties(Control control)
{
    super.setControlProperties(control);
    Button button = (Button)control;
    button.setEventHandler(Button.EVENT_ONCLICK,
        m_strOnClick, null);
    button.setDefault(m_bDefault.booleanValue());
    button.setWidth(m_strWidth);
    button.setHeight(m_strHeight);
    button.setLabel(m_strLabel);
}
private String m_bDefault = null;
...
```

Example 12-8. Setting default control values

You can set default control values in your call to `setControlProperties()`.

The following example sets a default string on a control:

```
if (m_strMyString != null)
{
    control.setMyString(m_strMyString);
}
else if ( control.getMyString() == null)
{
```

```

    control.setMyString("This is my default string");
}

```

Example 12-9. Setting the title on a label

The following example from a component class sets a title on a label control before the label is instantiated:

```

//Set a label from a form argument
public void onInit(ArgumentList arg)
{
    //Get the title argument
    //String strTitle = arg.get("title");

    //Pre-create the label control for the title
    Label lblTitle = getControl("title", Label.class);

    //Set the title property on the control class
    lblTitle.setValue(strTitle);
    ...
}

```



Caution: If you set an attribute on a control that you created during component initialization and then specify a setting for the same attribute in the control tag, the JSP tag value overwrites the initialized value.

Example 12-10. Setting a control label in the component class

In this example, the component class tests for a context value of user group and selects the label that is displayed on to the user based on the group value. To do this, get the control and call `setLabel()`, passing in the string value.

```

if (group.equals("HR") == true)
{
    Label lbl = (Label)getControl("attrLabel", Label.class);
    lbl.setLabel("Human Resources");
}

```

Example 12-11. Setting a control label in the tag class

You can set or get control values programmatically by calling methods defined in the tag class.

The following example, in a `render()` method, sets the value of a label:

```

import com.documentum.web.form.LabelTag;
...
protected void renderEnd(JspWriter out)
{
    LabelTag label = new LabelTag();
    String strValue = "My label";
    label.setLabel(strValue);
}

```

Example 12-12. Setting the state of a checkbox

The following example is an `onInit()` function in a JSP page that sets the named checkbox to unchecked:

```
<%
    public void onInit()
    {
        Checkbox cb = (Checkbox) getControl("selectAll", Checkbox.class);
        cb.setValue(false);
    }
%>
<dmf:row>
    <td>
        <dmf:checkbox name='selectAll' onclick='onSelectAll' />
    </td>
</dmf:row>
...

```

Getting datagrid controls

You can iterate through the rows of a datagrid and get values of the contained controls using `Control.getContainedControls()`.

Example 12-13. Getting a value from a datagrid row

The following example is taken from code that fetches a datafield value from the row, in order to test the value and render the row in a specific color depending on the datafield value. You must iterate through the rows to find the required control and value:

```
Datagrid grid = (datagrid) getControl("grid", Datagrid.class)
Iterator iterGrid = grid.getContainedControls();
while(iterGrid.hasNext())
{
    Control ctrl = (Control) iterGrid.next();
    if(ctrl instanceof DatagridRow)
    {
        Iterator iterRow = ctrl.getContainedControls();
        while(iterRow.hasNext())
        {
            Control ctrl = iterRow.
            if(ctrl instanceof Checkbox)
            {
                Checkbox chk = (Checkbox) ctrl;
                boolean isSelected = chk.getValue();
                if(isSelected)
                {
                    //test the value and set cssclass on the row
                }
            }
        }
    }
}

```

Passing arguments to action-enabled controls

You can pass arguments from the original query, such as a datagrid query, to the action precondition class.

Example 12-14. Passing a precondition argument

The CheckinAction precondition class takes an optional lock owner argument. If you pass this argument when the action control is rendered, then the precondition class will not have to look up the lock owner based on the object ID, which is a relatively slower process. You pass the precondition in an argument tag as follows:

```
<dmfx:actionmultiselect name="multiselect...>
<dmf:datagrid...>
  ...
  <datagridrow>
    <dmfx:actionmultiselectcheckbox name='multiselectcheckbox'>
      <dmfx:argument name='objectId' datafield='r_object_id'>
        <dmfx:argument name='lockOwner' datafield='r_lock_owner'
      </dmfx:actionmultiselectcheckbox>
    </datagridrow>
  </datagrid>
</dmf:datagrid>
</dmfx:actionmultiselect>

<dmfx:actionbutton action='checkin' dynamic='true'
  nlsid="MSG_CHECKIN"/>
```

The checkin action class CheckinAction gets the arguments as follows:

```
public boolean queryExecute(String strAction, IConfigElement config,
  ArgumentList arg, Context context, Component component)
{
  ...
  String strLockOwner = arg.get("lockOwner");
}
```

Programming databound controls

The following topics describe databound controls:

- [Data support classes, page 390](#)
- [Getting data, page 390](#)
- [Modifying the display and handling of attributes, page 395](#)
- [Rendering data with result sets, page 404](#)
- [Formatting data with handlers, page 406](#)
- [Adding custom attributes to a datagrid, page 407](#)

Data support classes

The databound control package `com.documentum.web.form.control.databound` contains controls and helper classes that read one or more values from a database or a repository and display the data in a formatted table or list. The data is retrieved from a DFC session interface, a JDBC connection, or an existing in-memory recordset. Other WDK controls can dynamically bind to the resulting data and display it.

The classes in the `com.documentum.web.form.control.databound` package can be grouped by the functions they perform:

- **DataProvider classes**
Base class and framework for any databound control to use to get data from a query or result set. (Refer to [Getting data, page 390.](#))
- **Databound controls**
Controls that bind to data from a DataProvider instance. Many of the controls in WDK can be databound, but they must have a DataProvider class that provides the query results or recordset. A databound control implements `IDataboundControl` or extends a class that implements it, for example, `DataDropDownList`, `DataListBox`, and `Datagrid`. Each databound control has a tag class that renders the layout for the databound control.
- **Data result set classes**
Classes that implement primitives for handling queried data, paged data, unbounded sets of data, and in-memory sets of data. (Refer to [Rendering data with result sets, page 404.](#))
- **Data handler classes**
Classes that format the data from result sets. (Refer to [Formatting data with handlers, page 406.](#))
- **Data binding utility controls and classes**
Classes that display data in various layouts and assist other framework classes. (Refer to [Control arguments, page 433.](#))

Getting data

The `DataProvider` class establishes the connection to the data source and reads the data received from the source. Other data manipulations such as paging, sorting, and caching, are handled by data handler classes. You can get the `DataProvider` class either in your component class or in the JSP page.

You can use a data provider to render data in a tag class. All tag classes that extend `ControlTag` can call `resolveDatafield()` method to get an instance of `DataProvider`.

You can also get data for your behavior class using `DfQuery`.

The following topics describe how to get data:

- [Getting data in a component, page 391](#)
- [Getting data in a tag class, page 391](#)
- [Getting data in a behavior class, page 392](#)
- [Getting or overriding data in a JSP page, page 393](#)
- [Refreshing data, page 394](#)
- [Caching data, page 394](#)

Getting data in a component

Example 12-15. Getting data in the component class

If your component JSP pages contain databound controls, you must instantiate a `DataProvider` class to provide data to the controls. The `setQuery()` method fetches the data. In the following example from `AclList`, the `DataProvider` class provides data:

```
public void onInit(ArgumentList args)
{
    super.onInit(args);

    // create (instantiate) the datagrid
    Datagrid datagrid = ((Datagrid) getControl(CONTROL_GRID, Datagrid.class));
    datagrid.getDataProvider().setDfSession(getDfSession());
    // set initial query for acls
    datagrid.getDataProvider().setQuery(BASE_QUERY + QUERY_ORDERBY);
    ...
}
```

Getting data in a tag class

For controls that are contained within a `datagrid`, the tag class can get the data from the enclosing `datagrid` using the `FindEnclosingDataProvider` utility class in the `com.documentum.web.form.control.databound` package.

Example 12-16. Getting data in a tag class

In the following example from `DataSortImageTag`, the `renderEnd()` method gets the enclosing `datagrid` control in order to access the sort columns:

```
DataSortImage sort = (DataSortImage) getControl();
Form form = getForm();
FindEnclosingDataProvider v = new FindEnclosingDataProvider();
```

```
sort.visitContainer(v);
DataProvider db = v.getDataProvider();
...
// find out if the parent dataprovider is sorted by this column
boolean bSorted = false;
String strCol = sort.getColumn();

// find out if we were the last column sorted
String strSortCol = db.getSortColumnName();
...
```

Getting data in a behavior class

Your component or action class may need to run a query that does not provide data directly to a databound control. In this case, use `DfQuery` to get the data.

A `java.sql.ResultSet` object can be obtained from XML definitions wrapped in a `ConfigResultSet`, from JDBC result sets, or from DFC `IDfCollection` objects wrapped in a `CollectionResultSet`.

Example 12-17. Running a query

The following example runs a query using `DfQuery`. The results are returned in an `IDfCollection`.

```
public IDfCollection getDocuments(IDfSession session,
    String additionalAttrs)
{
    IDfCollection result = null;
    String defaultAttrs = "r_object_id, object_name";

    // build up the query
    StringBuffer queryBuf = new StringBuffer(128);
    queryBuf.append("SELECT ")
        .append(" FROM dm_sysobject (ALL) WHERE r_object_id IN (");
    //appendStr() concatenates strings into a StringBuffer
    appendStr(m_documents, queryBuf);
    queryBuf.append(')');

    IDfQuery query = new DfQuery();
    query.setDQL(queryBuf.toString());
    result = query.execute(session, query.READ_QUERY);
    return result;
}
```

Note: When you use `IDfCollection` objects, you must close the collection after using it. Unclosed collections cause bugs that are hard to find.

In the component `onRender()` or `onInit()` method, you can fill a databound control with values from an XML file or JDBC result set.

Example 12-18. Setting control values from an XML file

The following example from AdvSearch reads elements from a search configuration file and stores it as a result set:

```
IConfigElement dateOptions = lookupElement(DATE_OPTIONS);
m_currentDateOptions = new ConfigResultSet(dateOptions.
    getDescendantElement(DATE_CONDITIONS), null, "label", "date",
    null, "value", "value");
```

Then the values from the XML configuration file are used to populate a dropdown list control:

```
DataDropDownList dateOptionList = (DataDropDownList) getControl(
    DATE_PARAMS, DataDropDownList.class);
dateOptionList.getDataProvider().setResultSet(
    m_currentDateOptions, null);
```

Getting or overriding data in a JSP page

There are two ways you can get data by modifying a JSP page:

- Override the databound control query attribute, if it has one
- Get the data provider and use it to get data

Example 12-19. Overriding a query in a JSP page

The following example sets a query for a datagrid in the JSP page:

```
<dmf:datagrid name="controlgrid1" cellspacing="2"
    cellpadding="3" bordersize="0" cssclass="databoundExampleDatagrid"
    paged="true" query="select r_object_id, object_name,
    home_url from km_enterprise">
</dmf:datagrid>
```

Example 12-20. Getting data in the JSP page

If your databound controls on the JSP page need access to a value in a query or recordset, you must include the DataProvider class and create an instance of it. In the following example from relationships_streamline.jsp, the data provider gets data for the datagrid control on the page:

```
<%
    Relationships form = (Relationships)pageContext.getAttribute(
        Relationships.FORM, PageContext.REQUEST_SCOPE);
    DataProvider dataProvider = ((Datagrid)form.getControl(
        Relationships.GRID_NAME, Datagrid.class)).getDataProvider();
%>
...
<%
    if (dataProvider.getResultsCount() > 1)
    {%>
```

```
...
<%
  }
%>
```

Refreshing data

If the data you display can be changed by another component, refresh the data using the data provider. For example, you can nest to the properties component from a datagrid. When the user changes a property, you must refresh when you return to the grid. The provider then requeries the data and steps through the results. It also reapplies sort settings.

Example 12-21. Refreshing data

The GroupWhereUsed class has an onRefreshData() method that uses a DataProvider instance to refresh data. The onRefreshData() lifecycle event handler is called by the form processor when a form is rendered:

```
public void onRefreshData()
{
    Datagrid datagrid = ((Datagrid)getControl(CONTROL_GRID, Datagrid.class));
    datagrid.getDataProvider().refresh();
}
```

Caching data

Caching is done automatically by the data handler classes. The default value of the cache is 100 rows. The DataProvider class sets the cache size based on the value in DataboundProperties.properties in the package com.documentum.web.form.control.databound (in /WEB-INF/classes). When the cache size is reached and more data is requested, the query is reissued to retrieve more results. The cache is discarded when the user navigates to another page.

You can override the cache size using DataProvider.setCacheSize.

Example 12-22. Overriding the record cache size

To override the cache size that is set in DataboundProperties.properties, call DataProvider.setCacheSize. In the following example from DatagridTag, the cache size is set to the value of the recordcachecount attribute on the datagrid tag:

```
if (m_nRecordCacheCount != null)
{
    grid.getDataProvider().setCacheSize(m_nRecordCacheCount.intValue());
}
```

You can change the cache size for an individual control by getting the control and calling `setCacheSize()`. You can also override the cache size by setting the value of the `recordcachecount` attribute on a `datagrid` tag.

Modifying the display and handling of attributes

You can modify how certain attributes or attribute types are displayed by using a formatter class. You can modify how the attribute is saved by using a value handler class. These classes are used to display or handle data in a `docbaseattribute`, `docbaseattributevalue`, or `docbaseattributelist` control.

You can modify which control is used to render an attribute by specifying a tag class that extends `DocbaseAttributeValueTag`. You can add configuration elements that are used by the formatter, handler, or tag class.

Custom formatters, handlers, and custom tag classes are registered in a `docbaseobject` configuration file whose root element is `<docbaseobjectconfiguration>`. A JSP page references this configuration by setting the `configid` attribute of `<dmfx:docbaseobject>` to the same value as the `id` of the `<docbaseobjectconfiguration>` element. If this attribute is not specified on the `docbaseobject` tag, the default configuration is used. This default configuration has an `id` of "attributes" and is found in `/webcomponent/config/library/docbaseobjectconfiguration_dm_sysobject.xml`.

The following topics describe the configuration file, formatters, value handlers, and `docbaseattributelist` lookup process:

- [docbaseobjectconfiguration file, page 395](#)
- [Attribute formatters, page 397](#)
- [Value handlers, page 398](#)
- [Tag classes, page 398](#)
- [Custom elements and editing components in object configuration, page 401](#)
- [Default configuration, page 402](#)
- [DocbaseAttributeList lookup process, page 403](#)

docbaseobjectconfiguration file

The `docbaseobjectconfiguration` definition contained in a configuration file with the primary element `<docbaseobjectconfiguration>` specifies all formatters and value handlers that should be applied for the display and handling of specific attributes and attribute types. You can also specify custom classes to display attributes that are generated by a `docbaseattributelist` control and a component that will be launched to

edit the attribute value. You can define additional elements to be used by your tag implementation.

Note: The formatters and handlers in this file apply to all instances of the attribute that are rendered by various DocbaseAttribute* controls. These controls are most commonly seen on the Properties/Attribute pages but are also used within Import and Check In screens. Components that retrieve their data through queries, such as navigation pages, do not render attributes based on this configuration file.

Specific attributes are listed in the <names> element, and attribute types are listed in the <types> element. The more specific attributes in the <names> element override the formatting or handling specified in the <types> element. For example, there is a formatter for attributes of the type "id" that displays the object name instead of the ID. This is overridden for the attribute r_object_id, whose formatter displays the actual ID as a text string.

The configuration file has the following elements:

```

1<docbaseobjectconfiguration id='attributes'>
2<names>
3  <attribute name='some_attribute">
4    <valuehandler>fully.qualified.class.name</valuehandler>
5    <valueformatter>fully.qualified.class.name</valueformatter>
6    <tagclass>fully.qualified.class.name</tagclass>
7    <labeltagclass>fully.qualified.class.name</labeltagclass>
8    <valuetagclass>fully.qualified.class.name</valuetagclass>
9    <editcomponent>fully.qualified.class.name</editcomponent>
10   <custom_element_name>some_value</custom_element_name>
</names>

<types>
11  <attribute type='type_name' repeatingonly="true | false"
    singleonly="true | false">
    <!-- Same elements as names.attribute above -->
</types>
</docbaseobjectconfiguration>

```

- 1** Defines formatters, value handlers, tags, editing components, and custom elements for object attributes or attribute types. The id attribute matches this configuration to the configid attribute on a docbaseobject control in a component JSP page.
- 2** Contains a list of attribute names for which there are special handlers.
- 3** Specifies the name for an attribute that is defined in the repository data dictionary.
- 4** Fully qualified class name for a class that implements IDocbaseAttributeSetValueHandler to set the value of the attribute. The handler class will be used by DocbaseAttribute and DocbaseAttributeValue to determine whether the attribute is modifiable. Use this class for modifiable attributes whose value cannot be set by the standard DocbaseObject.save() implementation, which uses the IDfTypedObject.setXXX methods.

- 5 Fully qualified class name for a class that implements `IDocbaseAttributeValueformatter` to format the display of the attribute. The formatter class will be used by `DocbaseAttributeValueTag` to determine the value that is rendered for display. Use this class for attributes whose value is not clear to the user. For example, the value for the attribute `r_resume_state` is an integer. The formatter class renders the name of the lifecycle state for display.
- 6 Fully qualified class name for a class that extends `DocbaseAttributeTag`. The class will be used by `DocbaseAttributeListTag` to render the attribute. Use this class to render the attribute when `<labeltagclass>` and `<valuetagclass>` are insufficient for the rendering requirements.
- 7 Fully qualified class name for a class that extends `DocbaseAttributeLabelTag`. The class will be used by `DocbaseAttributeListTag` to render the attribute label.
- 8 Fully qualified class name for a class that extends `DocbaseAttributeValueTag`. The class will be used by `DocbaseAttributeListTag` to render the attribute value. For example, this control could render a textarea instead of a text box for an attribute.
- 9 ID of component that will be used to display a UI for editing the attribute value. The component is launched from the **Edit** link that is rendered by `DocbaseAttributeValueTag`. The default is to use the `docbaserepeatingattribute` and `docbasesingleattribute` components for repeating and single attributes, respectively.
- 10 Custom elements can be inserted into the definition. The element and value will be available to the tag classes specified in the definition.
- 11 Specifies an attribute type in the data dictionary or a pseudotype that is handled by the tag class. The `repeatingonly` attribute on this element applies the customization only to multi-valued attributes. The `singleonly` attribute applies the customization only to single-valued attributes. The default value for both attributes is false.

Attribute formatters

Attribute formatters change the presentation of an attribute. Register your custom formatter for a specific attribute as the value of `<names><attribute name=...>.<attribute>.<valueformatter>`. Register your custom formatter for an attribute type as the value of `<types><attribute type=...>.<attribute>.<valueformatter>`.

The default presentation of an attribute is its value, but some attributes do not have values that are meaningful for the user. For example, the business policy (lifecycle)

ID has no meaning for most users, so the formatter looks up the policy name (error handling code removed):

```
public String getAttributeDisplayValue(String attribute,
    DocbaseObject docbaseObject)
{
    String value = null;
    IDfPersistentObject persistentObject = docbaseObject.getDfObject();
    IDfSysObject sysObject = (IDfSysObject)persistentObject;
    value = sysObject.getPolicyName();
    return value;
}
```

Value handlers

Attribute value handlers change the handling of a value. If the value cannot be saved using the standard `DocbaseObject.save()` method, or you need to perform additional processing after a save such as saving the value elsewhere in your application, implement a custom value handler class. Register your custom handler for a specific attribute as the value of `<names><attribute name=...>.<attribute>.<valuehandler>`. Register your custom handler for an attribute type as the value of `<types><attribute type=...>.<attribute>.<valuehandler>`.

The default presentation of an attribute is its value, but some values cannot be saved by the standard `DocbaseObject.save()` implementation. For example, the `r_version_label` attribute must be saved or deleted with a call to `IDfSysObject.mark()` or `unmark()`, respectively. The value handler class performs a check for read-only status and then calls `mark()` or `unmark()` in the `setAttributeValue()` method (error handling removed):

```
public void setAttributeValue(String attribute, IValue value,
    DocbaseObject docbaseObject) throws DfException
{
    IDfPersistentObject persistentObject = docbaseObject.getDfObject();
    ...
    String label;
    //test label
    IDfSysObject sysObject = (IDfSysObject)persistentObject;
    sysObject.unmark(labelBuffer.toString());
    //add new label
    String[] labelValues = value.getStringArray();
    sysObject.mark(label);
}
```

Tag classes

You can configure a custom tag class to render your attribute label, attribute value, or both. Your custom class will be instantiated by `DocbaseAttributeList` to render

the label and/or value. The following example adds a custom tag class that extends DocbaseAttributeValue to render the subject attribute of dm_document as a TextArea control instead of the default Text control. The default rendering of the subject attribute is a single line, as shown below:

Figure 12-1. String attribute rendered as text control

Title :	<input type="text" value="12-1-04 report"/>
Type :	dm_document
User Comments :	<input type="text"/>
Version Label :	1.1, CURRENT
Version Stamp :	
Version Tree Root Object :	
Virtual Document :	0
World Permissions :	<input type="text" value="None"/>

The following excerpt is from a copy of docbaseobjectconfiguration_dm_sysobject.xml, copied to /custom/config as docbaseobjectconfiguration_dm_document.xml:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<config version='1.0'>
<scope type="dm_document">
<docbaseobjectconfiguration id='attributes'>
  <names>
    <attribute name='subject'>
      <valuetagclass>com.mycompany.control.SubjectAttributeValueTag
      </valuetagclass>
      <lines>5</lines>
    </attribute>
    ...
  </names>
</docbaseobjectconfiguration>
</scope>
</config>
```

The elements in a docbaseobjectconfiguration file are described in detail in *Web Development Kit and Client Applications Development Guide*.

The custom tag class that is registered in the configuration file shown above extends DocbaseAttributeValue and sets the number of lines on the TextArea control from the custom element <lines/>:

```
package com.mycompany.control;

import com.documentum.web.formext.config.IConfigElement;
import com.documentum.web.formext.control.docbase.DocbaseAttributeValue;
import com.documentum.web.formext.control.docbase.DocbaseAttributeValueTag;
import com.documentum.web.formext.control.docbase.DocbaseObject;
import java.io.IOException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.JspWriter;

public class SubjectAttributeValueTag extends DocbaseAttributeValueTag
{
```

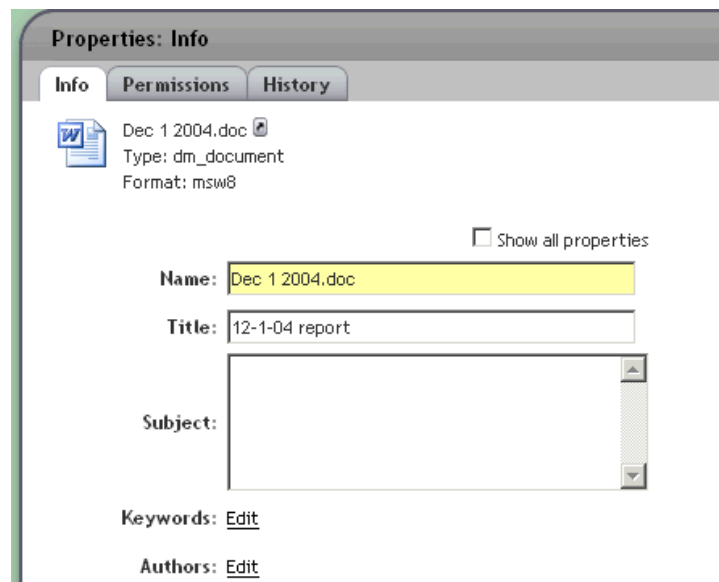
```
protected void renderSingleAttribute( String strFormattedValue,
                                     String strValue,
                                     boolean bReadOnly,
                                     boolean bHasCompleteList,
                                     JspWriter out)
    throws IOException, JspTagException
{
    //String strLines = "3";
    String strLines = setDynamicLines();
    DocbaseAttributeValue value = (DocbaseAttributeValue) getControl();
    value.setLines(strLines);
    super.renderSingleAttribute(strFormattedValue, strValue,
                               bReadOnly, bHasCompleteList, out);
}

private String setDynamicLines()
{
    DocbaseAttributeValue value = (DocbaseAttributeValue) getControl();
    DocbaseObject obj = (DocbaseObject) getForm().getControl(value.getObject());

    IConfigElement iConfigElement = obj.getConfigForAttribute(
        value.getAttribute(), "lines");
    if (iConfigElement != null)
    {
        return (iConfigElement.getValue());
    }
    return "1";
}
}
```

After restarting the application server, the same attribute is rendered as a `TextArea` control as shown below:

Figure 12-2. String attribute rendered as TextArea control



Custom elements and editing components in object configuration

You can specify a component that will be launched to edit the specified attribute (`<attribute name=...>` or attributes of the specified type (`<attribute type=...>`). For example, the default `docbaseobjectconfiguration` definition specifies that the `versionlabels` component should be used to edit the `r_version_label` attribute.

You can add elements whose values will be used in your custom tag class. For example, the `rich_text` pseudoattribute specifies custom tag classes and custom elements such as `<showfonts>`. The boolean value of the `showfonts` element is used in the `RichTextDocbaseAttributeValueTag` class:

```
DocbaseAttributeValue value = (DocbaseAttributeValue) getControl();
DocbaseObject obj = (DocbaseObject) getForm().getControl(value.getObject());

IConfigElement iConfigElement = obj.getConfigForAttribute(value.getAttribute(),
    "showfonts");
if (iConfigElement != null)
{
    richtext.setHasFonts(iConfigElement.getValue());
}
```

Default configuration

If no configuration is specified on the DocbaseObject control, then the default configuration in `docbaseobjectconfiguration_dm_sysobject.xml` is used. The following attributes have special handling as specified in this configuration file:

Table 12-3. Default attribute handling

Attribute	Customization
<code>a_storage_type</code>	Renders storage name instead of integer
<code>r_object_id</code>	Renders object ID as text, overrides the rendering of the ID type
<code>i_chronicle_id</code>	Renders chronicle ID as text, overrides the rendering of the ID type
<code>r_policy_id</code>	Renders policy name instead of ID
<code>r_version_label</code>	Sets changes to version labels, uses <code>versionlabels</code> component instead of <code>docbaserepeatingattributes</code> component
<code>r_current_state</code>	Renders state name instead of state number
<code>r_resume_state</code>	Renders state name instead of state number
<code>type="id"</code>	Renders name or path instead of ID value. Overridden by specific attribute customizations. To remove specific customizations, or this type-based customization, comment out the attribute element in your extended configuration.
<code>type="rich_text"</code>	Specifies handlers for attributes with the pseudoattribute type "rich_text." This customization is applied to attributes in a scoped <code>attributelist</code> definition that contains in <code><pseudo_attributes></code> element. The attribute within this element must have the type <code>rich_text</code> , for example: <code><attribute name="folder_description" type="rich_text" ...></code>

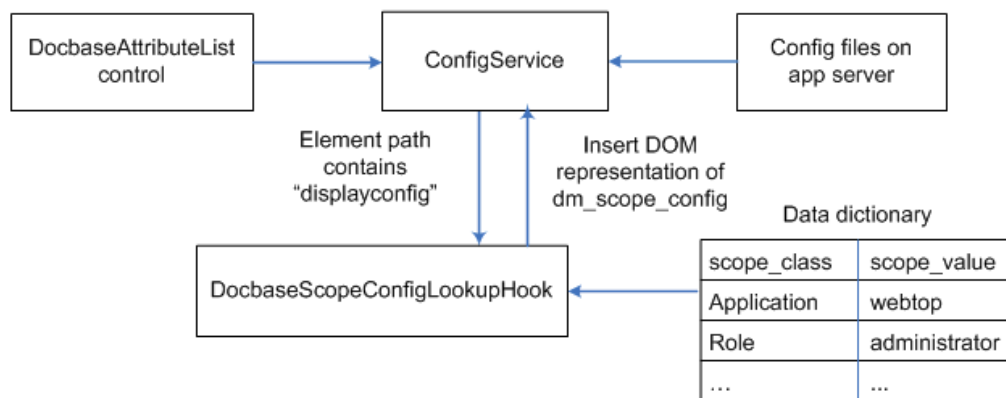
DocbaseAttributeList lookup process

WDK registers a lookup hook to intercept lookup calls from the `docbaseattributelist` control. The hook class `DocbaseScopeConfigLookupHook` is registered in `com.documentum.web.formext.Environment.properties`. The lookup hook has the following settings:

```
LookupHookPath.1=displayconfig.*
LookupHookArgument.1=relative
LookupHookClass.1=com.documentum.web.formext.config.DocbaseScopeConfigLookupHook
```

`DocbaseAttributeList` class looks up the element `displayconfig.root`, and the lookup hook then collects the display configuration information from the data dictionary or config file, depending on whether the `docbaseattributelist` configuration is set to use data dictionary lookup (`<data_dictionary_population>`). This process is diagrammed below:

Figure 12-3. Docbaseattributelist lookup



`DocbaseScopeConfigService` collects the display configuration information for the current repository from its data dictionary and passes a DOM representation of the `dm_scope_config` object to `ConfigServices`. `DocbaseScopeConfigService` also keeps a map of repositories, locales, and data dictionary application names.

When a `DocbaseAttributeValue` control is rendered by `DocbaseAttributeValueTag`, it looks up the repository to get associated value assistance for the attribute by calling `DocbaseAttributeValue::hasValueAssistance`. The appropriate value assistance controls are rendered for value assistance: dropdown, label, or link. For attributes with conditional value assistance, an event handler is registered to do preprocessing.

Rendering data with result sets

A data provider returns data in the form of a result set. You must select the appropriate result set class to handle the data. The `DataProvider` class matches a data handler class to the result set in order to render the data.

You must set the result set for the databound control using `DataProvider.setResultSet()` (refer to below).

The result sets are described in the following topics:

- [Making data scrollable, page 404](#)
- [Handling data from a configuration file, page 404](#)
- [Handling data from an array or vector, page 405](#)

Making data scrollable

The base recordset class, `ScrollableResultSet`, creates in-memory recordset objects. `ScrollableResultSet` extends `java.sql.ResultSet`. All other in-memory recordset objects, such as those backed by arrays or lists, extend `ScrollableResultSet`. Each type of scrollable result set handles a different type of data: You can pass data from a scrollable result set to a data grid.

You can call `setScrollableResultSet()` on `DataProvider` to set data in a databound control.

Example 12-23. Setting data to a databound control

In the following example from `ChangeHomeDocase`, a `ListResultSet` is set by the `DataProvider` to a data dropdown list control:

```
DataDropDownList docbaseList = (DataDropDownList) getControl(
    DOCBASE_LIST, DataDropDownList.class);
docbaseList.getDataProvider().setDfSession(getDfSession());

ListResultSet docbaseListResultSet = new ListResultSet(
    getDocbaseNames(), "docbasename");
...
docbaseList.getDataProvider().setScrollableResultSet(docbaseListResultSet);
```

Handling data from a configuration file

`ConfigResultSet` gets data from an XML configuration file. Refer to [Passing Data from a Vector to a TableResultSet, page 405](#) for an example of putting configuration file content into a `Vector` and passing the data to a table or list.

Example 12-24. Passing data from a configuration file to a data dropdown list

In the following example from AdvSearch, the ConfigResultSet reads the value of the config file element <date_options> and its child element <date_conditions> and the values to a datadropdownlist control:

```
IConfigElement dateOptions = lookupElement(date_options);
m_currentDateOptions = new ConfigResultSet(dateOptions.
    getDescendantElement(date_conditions), null, "label",
    "date", null, "value", "value");

//put the results into the named control
DataDropDownList dateOptionList =
(DateDropDownList)getControl(dateparams, DataDropDownList.class);
dateOptionList.getDataProvider().setResultSet(m_currentDateOptions, null);
```

Handling data from an array or vector

Example 12-25. Handling a vector of data in ArrayResultSet

In the following example from ChangePassword, Array data is passed to an ArrayResultSet data handler and then to a ScrollableResultSet:

```
Collator collator = Collator.getInstance(LocaleService.getLocale());
// do case-insensitive comparison
collator.setStrength(collator.SECONDARY);
Arrays.sort(docbases, collator);
ArrayResultSet arrDocbases = new ArrayResultSet(docbases, "docbase");
arrDocbases.sort("docbase", IDataboundParams.SORTDIR_FORWARD,
    IDataboundParams.SORTMODE_TEXT);
listDocbases.getDataProvider().setScrollableResultSet(arrDocbases);
```

The TableResultSet handles a vector of data and returns it in table format.

Example 12-26. Passing Data from a Vector to a TableResultSet

In the following example from Administration, the Vector consists of data read from elements in the configuration file. The Vector is passed to a table result set which is displayed in a datagrid:

```
Vector oToolData = new Vector();
//add config data to Vector
...
m_oAdminTools = new TableResultSet(oToolData, s_strDefaultAttributes);
((Datagrid)getControl(CONTROL_GRID)).getDataProvider().setScrollableResultSet(
    m_oAdminTools);
```

The ListResultSet handles a vector of data and returns it as a list.

Example 12-27. Passing Data from a Vector to a ListResultSet

In the following example from UserImport, getDocbaseNames() provides a Vector of repository names for dropdown list:

```
protected Vector getDocbaseNames()
{
    Vector docbaseNames = new Vector();
    IDfDocbaseMap docbaseMap = client.getDocbaseMap();
    //iterate through Map from DocBroker
    return docbaseNames;
}
...
DataDropDownList docbaseList = (DataDropDownList) getControl(
    USER_HOME_LIST,DataDropDownList.class);
ListResultSet docbaseListResultSet = new ListResultSet(
    getDocbaseNames(),"docbasename");
docbaseListResultSet.sort("docbasename", IDataboundParams.
    SORTDIR_FORWARD, IDataboundParams.SORTMODE_TEXT);
docbaseList.getDataProvider().setScrollableResultSet(docbaseListResultSet);
```

Formatting data with handlers

Data handlers are used by the `DataProvider` class to format the data from result sets. Each data handler is paired with a result set. For information on custom attribute data handlers, refer to [Adding custom attributes to a datagrid, page 407](#).

DataHandler — Abstract data handler class.

QueryDataHandler — Used with the results of a query. The handler provides cached access to the `RecordSet` generated by the query. You can set the default cache size (number of rows) in your Web application in the resource file `DataboundProperties.properties` found in `/WEB-INF/com/documentum/web/form/control/databound/`.

Sorting: When the data is sorted, a new query is performed using the column sort information.

Caching: Records are cached in memory up to the size of the cache or up to the current page required by the user. If the user requests a page not currently in the cache, the data is queried and another data block is cached. `ResultSet` objects are never left open beyond the life of a page render.

Performance: In large result sets, there is a performance hit to query and reopen the `ResultSet`.

ResultSetDataHandler — This data handler is used when a `ResultSet` and a `Statement` are used. The data handler provides access to data from an existing `ResultSet`.

Sorting: Callbacks must be handled by the developer.

Caching: Paging is provided.

ScrollableResultSetDataHandler — This data handler is used with a class that extends `ScrollableResultSet`. The data handler provides access to data held in a scrollable `ResultSet` object.

Sorting: Callbacks must be handled by the developer of the `ScrollableResultSet` class.

Caching: Paging is provided.

`DFCQueryDataHandler`: This data handler is used when a DQL string is provided, and the query is executed using an `IDfQuery` object.

Adding custom attributes to a datagrid

To add columns of data to a datagrid, you can define a column in the component definition and render the column with a formatter. Each formatter queries the individual values, which can slow performance. To improve performance, you can display custom attributes using a custom attribute data handler that implements `ICustomAttributeDataHandler`. This data handler can add an entire column of data to the underlying record set when a datagrid is rendered.

The custom attribute data handler serves the following purposes:

- Adds columns of data to a grid where the data comes from other queries
- Supplies hidden columns of data to actions whose preconditions require data retrieved from other queries

The custom attribute data handler and record set — A custom attribute data handler class implements `getRequiredAttributes()`, which is called by the `DFCQueryDataHandler` class to determine which attributes are required by the data handler. The custom handler implements `getData()`, which is called if the custom attribute columns that are specified in the XML definition are present in the underlying record set. The method `getData()` fills a custom record set with appropriate values.

The custom attribute handler is paired with a custom record set that implements `ICustomAttributeRecordSet`.

WDK provides a custom attribute data handler, `DFCQueryCustomAttributeDataHandler`, which will get data for custom attributes if they are specified in a column element of a component definition. If the object type does not have the attribute, the column is empty for that object. For example, if your custom drilldown component definition contains attribute columns for a custom type, objects of that type will be displayed with their custom attribute values. Objects in the list that do not have those custom attributes will be displayed with empty values in each custom attribute column.

To create a custom attribute data handler:

If your attribute display requires special processing before displaying the column, create a custom attribute data handler.

1. Implement `ICustomAttributeDataHandler` (refer to below) to retrieve the custom parameter.
2. Add an entry for the custom data handler in `app.xml`. For example:

```

...
<application>
  <custom_attribute_data_handlers>
    <custom_attribute_data_handler>fully_qualified_class_name
    </custom_attribute_data_handler>
  </custom_attribute_data_handlers>
  ...
</application>

```

3. Add the custom parameter to your action definition to pass it from the JSP page to the precondition class, action class, or component that is launched by the component.
4. Add the custom parameter to the control instance on the form.
5. Modify the action precondition or execution class, or the component class, to use the new parameter.
6. Add the custom parameter to the component column configuration.
7. If your component's controls require invisible parameters, include the `<loadinvisibleattribute>` element in the `<columns>` definition.

Note: Some components, such as `com.web.component.library.search.Search`, override `ComponentColumnDescriptorList` and pass all columns to the `DataProvider` instance. You must override this behavior and change the UI to hide the attributes that should be invisible to the user.

Example 12-28. Custom attribute data handlers in Digital Asset Manager

The Digital Asset Manager (DAM) application uses several custom attribute data handlers that provide examples of custom data handlers that pass hidden data to actions. The DAM custom attributes are hidden from the data display and passed to actions when the user selects objects in the display. Several custom data handlers are defined in the `dam app.xml` file, for example:

```

<custom_attribute_data_handler>
  com.documentum.dam.formext.docbase.TotalStoryboardsAttributeHandler
</custom_attribute_data_handler>

```

Custom attributes are specified in the component XML file as columns. The following example is excerpted from `dam_myfiles_classic_component.xml` and specifies an invisible column of data for total number of storyboards. Note that the `loadinvisibleattribute` element must have a value of `true` in order for the invisible columns to be queried:

```

<columns>

```



```

    <loadinvisibleattribute>true</loadinvisibleattribute>
    ...
    <column>
        <attribute>total_storyboards</attribute>
        <visible>>false</visible>
    </column>
</columns>

```

The data handler class `TotalStoryboardsAttributeHandler` implements `getRequiredAttributes()` to require the object IDs:

```

public String[] getRequiredAttributes()
{
    return new String[] {"r_object_id"};
}

```

The data handler class implements `getData()`, passing in the `ICustomAttributeRecordSet` to receive the query data:

```

public void getData(IDfSession dfSession, ICustomAttributeRecordSet recordSet)
{ ... }

```

The `getData()` method then gets the object IDs and puts them into an `ArrayList`, in preparation for building the query:

```

String[] objectIdArray = recordSet.getAttributeValues("r_object_id");
List objectIds = new ArrayList(Arrays.asList(objectIdArray));

```

Next (code not shown) the method builds and executes a query that will get the number of storyboards for each object ID. The method then iterates over the returned `IDfCollection` to set the data in the recordset:

```

recordSet.setCustomAttributeValue(iRow, s_attributeName, strTotalStoryboards);

```

The `total_storyboards` datafield is passed as an argument by the datagrid to two controls in the `myobjects_drilldown_body` JSP page: an `actionimage` and an `actionlinklist`. Thus the hidden datafield value is passed to the actions represented by the `actionimage` and `actionlinklist`.

Generating UI

The JSP tag classes provide methods to write out HTML and JavaScript. You can also override these methods and call other methods that write out data.

Each control tag generates one or more HTML elements. The HTML element name is formed from the form name and the control name, the control index, and an optional ID.

```

formname_controlname_controlindex

```

Example 12-29. Generating HTML

The following example from a JSP tag class generates HTML to the browser:

```
protected void renderEnd(JspWriter out)
    throws IOException
{
    MyControl myControl = (MyControl)getControl();
    StringBuffer buf = new StringBuffer(256);
    .append(myControl.getElementName()).append(" ");
    if (myControl.getId() != null)
    {
        buf.append(" id=").append(myControl.getId()).append(" ");
    }
    buf.append(" value=")
    .append(formatText(myControl.getValue()))
    .append(">");
    out.println(buf.toString());
}
```

Example 12-30. Setting a control value in onRender()

The following example sets a label with the value of the selected object ID in a component `onRender()` method. You must set the control state before calling `super.onRender()`:

```
public void onRender()
{
    IDfId objId = new DfId(objectId);
    try
    {
        dmAdminSession = getDmAdminSession();
        IDfSysObject obj = (IDfSysObject)dmAdminSession.getObject(objId);
        Label objectIdLabel = (Label)getControl("objectIdLabel",Label.class);
        objectIdLabel.setLabel(objId);
        Label objectNameLabel = (Label)getControl("objectNameLabel",Label.class);
        objectNameLabel.setLabel(obj.getObjectName());
        Label dumpLabel = (Label)getControl("dump", Label.class);
        dumpLabel.setLabel(obj.dump());
    }
    catch (DfException e)
    {
        e.printStackTrace();
    }
    super.onRender();
}
```

Generating a link in a control

Links that are generated by controls are susceptible to server errors in a high latency network. The errors occur when users are waiting for a page to refresh and click on several links. The browser cancels the first request and processes only the last click. The browser kills the connection associated with the early links, which are still being processed by the server. Since the connection to early links was closed by the browser,

the server code generates exceptions: socket exceptions and temporary instability including incorrectly generated pages.

To avoid this problem, you should not put a URL or JavaScript call into the HREF tag but instead use the onclick handler to call `postServerEvent()`, which implements a client-side locking. Put the number symbol "#" into the HREF so that the link will appear active. The actual action is done by the onclick event handler that is specified in the HTML tag.

Example 12-31. Rendering an HTML link

The `ControlTag` class renders a link using the `renderEventHREF()` method. Your custom control tag should call `renderEventHREF()` to render the link. In the following example from the `renderEnd()` method of `DataSortLinkTag`, the link is rendered into HTML

```
buf.append("<a");
renderEventHREF(buf, DataSortLink.EVENT_ONCLICK);
```

This method will generate a link similar to the following HTML:

```
<a href="#" onclick='postServerEvent(...);' ...>Sort</a>
```

Note: `setKeys(event)` always returns false. The onclick handler must return false to prevent the processing of the HREF content.

The `postServerEvent()` call ensures that only one click will be processed by the browser at a time. The first click will be processed, and subsequent clicks will be ignored until the first click has been completely processed.

Making a control accessible to JavaScript

The method `getFunctionName()` is a utility method in the `Control` class that is used in a control tag class to generate a function name including a unique hex number to use for generated JavaScript. For example:

Example 12-32. Generating a function name in JavaScript

```
getFunctionName()=>_x0
//or
getFunctionName("onclick")->_x0onclick
```

In the following example, the `FileBrowse` tag class generates a hidden input control:

```
String strUpdateHiddenCtrlFunctionName = filebrowse.getFunctionName();
out.write("<input type='hidden'");
out.write(" name='");
out.write(strHiddenCtrlName);
out.write("' id='");
out.write(strHiddenCtrlName);
out.write('\''');
out.write(" value='");
out.write(formatAttribute(filebrowse.getValue()));
out.write('\''');
```

```
out.write('>');
```

Displaying folder paths and breadcrumbs

You can display the user's navigation path in a component using the `primaryfolderpathlink` control. Primary folder paths are also displayed in folder links and attributes.

Subscriptions store the navigation folder path so that the user will see the same path when the subscription is displayed.

The following topics describe the interfaces for folder path and breadcrumb support:

- [Getting the primary folder path, page 412](#)
- [Displaying the folder path, page 412](#)
- [Adding support for a breadcrumb, page 413](#)
- [Using a hidden folder path in a component, page 414](#)

Getting the primary folder path

An object's primary folder path is calculated by the WDK application for each user from the list of possible folder paths (`r_folder_path` entries) to an object: the first path on which the user has browse permissions on every folder in the path is taken as the primary folder path for that user. Thus the primary folder path to an object for one user may be different from the primary folder path for another user.

To get the object's primary folder path for the user, call `FolderUtil.getPrimaryFolderPath()`, passing in an object ID and optionally a boolean flag that specifies whether the name of the passed object should be appended on the return path.

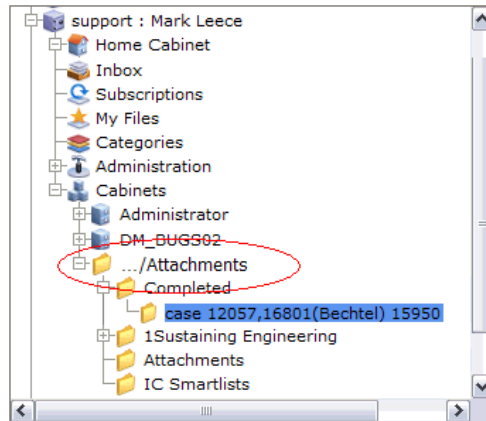
If your component can list the `r_folder_path` attribute of objects, add a cell template to the JSP page that uses a `primaryfolderpathlink` or `FolderUtil.formatFolderPath()` to display the value.

Displaying the folder path

If the user does not have browse permissions on any full path, the first entry in the list of paths is used as the primary folder path for display, and the folders on which the user does not have browse permissions are displayed as a partial folder path with ellipses. For example, if the user has browse permissions on the Documents cabinet and the

Attachments folder in the path /Documents/Reviews/Attachments, the path will be displayed to the user as follows:

Figure 12-4. Folder path display



In a partial folder path such as the example above, only the portion on which the user has full browse permissions is displayed. The strings for path display are generated by `FolderUtil.formatFolderPath(String strFolderPath)`. The return values are illustrated in the table below. The user has access to the underlined folders:

Table 12-4. Folder path display rules

Argument value	Return value
<u>"/a/b/c/d/e"</u>	<u>"/a/b/c/d/e"</u>
"/a/b/c/d/e"	<u>"/.../c/d/e"</u>
<u>"/a/b/c/d/e"</u>	<u>"/.../d/e"</u>
"/a/b/c/d/e"	" "
"/" or "/"	"/"
" " or null	" "

Adding support for a breadcrumb

You can use a breadcrumb for components by adding the breadcrumb control to the component JSP page. Components that extend `ObjectLocator` need no additional support for the breadcrumb.

If the component that displays a breadcrumb is in a container, add the breadcrumb to the container JSP page. If the contained component does not extend `ObjectLocator`, your container class must implement `IBreadcrumbContainer`. (Refer to the source code for `LocatorContainer` for an example.) Implement `getBreadcrumbPath()` and `setBreadcrumbPath()` so that you get and set the breadcrumb path or event handler on the control that you get from the container. The contained component should handle the breadcrumb event that is specified in `Breadcrumb.setEventHandler()` call. For example, if you have the following call in your contained component, you will need to implement a method called `"onClickBreadcrumbABC()"`:

```
breadcrumb.setEventHandler(  
    Breadcrumb.EVENT_ONCLICK, "onClickBreadcrumbABC", this);
```

Initialize a breadcrumb control by passing a prefix, folder path, and postfix to `FolderUtil.initBreadcrumb()`.

The full or partial folder path that is returned by `getPrimaryFolderPath(strObject)`, and the folder path formatting that is returned by `formatFolderPath()`, are cached for 10 minutes to improve performance of repeated calls. You can refresh the cache after folder properties or permissions have changed or a folder has been linked or unlinked by calling `FolderUtil.refresh()`.

Using a hidden folder path in a component

Navigation components pass repository folder paths in a `dmf:hidden` control and use the value to support the browser **Back** button.

Components that write repository folder paths in hidden controls can encrypt the value using the `encrypt` attribute so that names of folders the user does not have rights to view are not written in the generated HTML hidden field. For example:

```
<dmf:hidden name='folder-path' encrypt='true'/>
```

Get the path in the component class as shown in the following example:

```
public void onClickFolder(Link linkbtn, ArgumentList args)  
{  
    // get current path  
    String strPath;  
    Hidden pathCtrl = (Hidden)getControl(CONTROL_FOLDERPATH, Hidden.class);  
    if (pathCtrl.getValue() != null)  
    {  
        StringBuffer buf = new StringBuffer(  
            pathCtrl.getValue().length() + linkbtn.getLabel().length() + 1);  
        buf.append(pathCtrl.getValue())  
            .append('/')  
            .append(linkbtn.getLabel());  
        strPath = buf.toString();  
        ...  
    }  
}
```

```

    updateContextFromPath(strPath);
}

```

Note: Any hidden control value can be encrypted to hide it from the **View Source** browser feature. When your component class calls `getValue()` on the Hidden control class instance, the value is decrypted.

Implementing multiple selection

Web applications can support actions on more than one selected object. The `actionmultiselect` and `actionmultiselectcheckbox` controls provide support for invoking actions on multiple selected items. Use the `actionmultiselectall` control within a header to allow the user to select all checkboxes with a single click.

Note: When the user accessibility option is turned on, multiple selection is replaced by a link to an action page for each item and a global actions link.

The `actionmultiselect` control contains a block of `actionmultiselectcheckbox` controls, typically within a `datagrid`. In the following example, a checkbox is rendered next to each item, identified here by the `<dmf:argument>` tag with a datafield that is populated by `r_object_id`:

```

<dmfx:actionmultiselect name='multiselect'>
  <dmf:datagrid ...>
    <datagridrow>
      <dmfx:actionmultiselectcheckbox name='multiselectcheckbox'>
        <dmf:argument name='objectId' datafield='r_object_id' />
      </dmfx:actionmultiselectcheckbox>
      ...
    </datagridrow>
    ...
  </dmf:datagrid>
</dmfx:actionmultiselect>

```

Within each `actionmultiselectcheckbox` control, you can embed argument controls to pass arguments to the associated dynamic actions and to define the context that will be used to resolved the appropriate action definition. The actual action to be performed is defined using standard action controls, with the dynamic attribute set to `multiselect`. These action tags can be located in other frames.

Note: You can use only one `actionmultiselect` control per set of open frames in your application. If you have more than one `<actionmultiselect>` tag, then dynamic controls do not know which selected item to operate on. You cannot embed an `actionmultiselect` control within another `actionmultiselect` control.



Caution: The states of all actions associated with dynamic action controls are evaluated when the actionmultiselect control is rendered. A large number of selectable items or associated actions can degrade performance. For example, if there are ten selectable items and a hundred associated actions, one thousand states will be evaluated.

There are two ways you can get the arguments for multiple selections:

- Use `MultiArgumentDialogContainer` for your component

For example, the `SendToDistributionList` component uses this container.

The `MultiArgumentDialogContainer` passes the arguments to one instance of the `SendToDistributionList` component for each selected object.

- Add a required `componentArgs` parameter to your container:

```
<param name="componentArgs" required="true"></param>
```

In your container `OnInit()` method, get the arguments and pass them to the contained component in the following way:

```
//set array of component arguments
String strComponentArgs[] = arg.getValues("componentArgs");
ArgumentList componentArgs = new ArgumentList();
for (int i=0; i < srgComponentArgs.length; i++)
{
    String strEncodedArgs = strComponentArgs[i];
    componentArgs.add(ArgumentList.decode(strEncodedArgs));
}
setContainedComponentArgs(componentArgs);
```

Retrieve the argument collection in your contained component:

```
String [] vals = arg.getValues("objectId");
```

Managing control events

To fire a client event from a client control, follow the scripting language rules for firing and handling client events. If you wish to register a particular event handler for the client event, refer to [Registering client event handlers, page 114](#).

Server events that are handled on the server are implemented as calls to methods in server classes. Follow the Java language rules by importing and calling the appropriate class method.

The following topics describe control event handling on the server and some typical event scenarios in custom Web applications:

- [Use server-side or client-side processing?, page 417](#)
- [Firing a server event from the client, page 417](#)
- [Handling a control event on the server, page 420](#)
- [Updating components with client events, page 421](#)

- [Firing a client event from the server, page 422](#)
- [Linking controls by events, page 422](#)
- [State change events, page 423](#)
- [How control events are raised, page 424](#)
- [Using modal windows, page 425](#)
- [Setting event handlers programmatically, page 427](#)
- [Control lifecycle events, page 428](#)

Use server-side or client-side processing?

When you develop a WDK component, answer the following questions to help you choose between client-side and server-side event handling:

Is the cost of a client/server round trip too expensive for the user interaction? For example, is the user connected over a narrow-bandwidth line? If yes, then a client-side event handler may be more appropriate.

Is a highly dynamic user interface required? If yes, then a client-side event handler may be more efficient.

Are calls to business logic required? If yes, then a server-side event handler is required.

Firing a server event from the client

Any server-side event handler may be called from the client. Server-side event handlers are automatically called by the framework for control events unless the `runatclient` attribute is set to `true`.

postServerEvent — The WDK event client script `events.js` provides a `postServerEvent` function that you can use to explicitly call a server event handler. This script is automatically included in all rendered forms. The signature of the `postServerEvent` is:

```
function postServerEvent(strFormId, strSrcCtrl, strHandlerCtrl,  
    strHandlerMethod, strEventArgsName, strEventArgsValue);
```

where:

- `strFormId`: String ID of the form to submit. If null, the first form on the page is assumed.
- `strSrcCtrl`: String ID of control that fires the event (optional).
- `strHandlerCtrl`: String ID of the control that handles the event (optional).

- `strHandlerMethod`: String Java method name of the event handler in a class on the J2EE application server.
- `strEventArgName`: String Event argument name
- `strEventArgValue`: String Event argument value

Note: Note: Multiple argument names and values may be specified.

You can generate a `postServerEvent` call using the `<dmf:postserverevent>` tag, whose attributes supply the arguments to the function. Use this tag to generate the `postServerEvent()` call within a portlet JSP page. You can pass only one event argument and value with this control.

You can also call the `postServerEvent` function explicitly. When you explicitly use the `postServerEvent()` function, the first three arguments are typically passed as `NULL`. However, if your JSP page is in a portlet, you must have a named form, so you should provide a form name for the first parameter.

Typically, `postServerEvent` is called from within a client-side event handler. For example:

```
<script>
function handleClick(srcObject)
{
    postServerEvent(null, null, null, "onUpdateData", "objectId", id,
        "type", type);
}
</script>
```

An arbitrary number of event arguments and values may be passed to the server-side event handler by an explicit `postServerEvent()` call. The `dmf:postserverevent` tag passes only one event argument and value.

In the server-side event handler, the event arguments are available in the usual manner through the second `ArgumentList` parameter. For example:

```
public void eventHandler(Control control, ArgumentList arg)
{
    String argValue1 = arg.get("argName1");
    String argValue2 = arg.get("argName2");
}
```

When `postServerEvent` is called, the generated HTML Form is submitted (via an HTTP POST) to the J2EE server. The Form Processor accepts the request and invokes the named server-side event handler method on the Form. You must ensure that the JSP page imports the Java class that contains the named event handler.

The `postServerEvent()` method ties control events to their associated server-side event handlers. Each WDK control generates the appropriate HTML and JavaScript to hook the call to `postServerEvent()`. The generated HTML for the Button Control is shown below:

```
<input type='button' name='__10_btn' value='Update Status - Available'
    class="defaultButtonHtmlStyle" onclick='postServerEvent
    "__1013087507570_1", "", "__1013087507570_1", "onUpdateStatus");'>
```

The button control generates the HTML and JavaScript when the form is rendered. The button tag class renders the standard HTML `<input>` tag and associated onclick event so that when the user clicks on the HTML button, the `postServerEvent` is invoked. To ensure that the appropriate server-side event handler is located when the request is sent back to the server, the button control populates the first three arguments of `postServerEvent`: The ID of the form, the ID of the firing control, and the ID of the handler control.

Example 12-33. Client to server event

The following JSP page and associated Java behavior class demonstrate the use of client-side control events that are posted as server-side events.

The layout consists of two buttons, each with a status argument. Each button event updates the form layout to show the status of the selected button. The status is displayed using a WDK label control and is updated using a server-side event handler called `onUpdateStatus()`. This event handler accepts as an argument the status string to be displayed. The onclick event of the first button is handled normally on the server. The onclick event of the second button is handled on the client-side by setting its `runatclient` attribute to `true`.

A JavaScript event handler in the layout JSP page accepts a status argument in the same manner as its server-side counterpart. The event handler modifies the status by adding the message "intercepted by client" and then uses `postServerEvent` to call the server-side `onUpdateStatus` event handler with the modified status. The server-side event handler updates the status label as before and the form is redisplayed with the updated status.

When the first button is pressed, the form displays the status "Available". When the second button is pressed, the form displays the status "Off-line (intercepted by client)". This demonstrates the addition of the message string by the client event handler to the argument value for the second button ("Off-line").

JSP client event —

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="/WEB-INF/tlds/dmform_1_0.tld" prefix="dmf" %>
<%@ taglib uri="/WEB-INF/tlds/dmformext_1_0.tld" prefix="dmfx" %>
<html>
<head>
<dmf:webform formclass="com.documentum.web.samples.client.PostServerEvent"/>
<title><dmf:label label="Simple postServerEvent Example"/></title>
<script>
  function onUpdateStatus(obj, status)
  {
    var newStatus = status + " (intercepted by client)";
    alert("Client-side setting status from '" + status + "' to '" +
      newStatus + "'");
    postServerEvent(null, null, null, "onUpdateStatus", "status",
      newStatus);
  }
</script>
</head>
```

```
<body class='contentBackground'>
<dmf:form>
<h3><dmf:label label="Simple postServerEvent Example"/></h3>

<dmf:button label="Update Status - Available" onclick="onUpdateStatus">
  <dmf:argument name='status' value='Available' />
</dmf:button>
<br>
<br>
<dmf:button label="Update Status - Off-line" onclick="onUpdateStatus"
  runatclient="true">
  <dmf:argument name='status' value='Off-line' />
</dmf:button>
<br>
<br>Status: <dmf:label name="status"/>
</dmf:form>
</body>
</html>
```

Server event handler —

```
package com.documentum.web.samples.client;
import com.documentum.web.form.Form;
import com.documentum.web.form.control.Button;
import com.documentum.web.form.control.Label;
import com.documentum.web.common.ArgumentList;
public class PostServerEvent extends Form
{
  /**
   * Handle server-side event
   */
  public void onUpdateStatus(Button button, ArgumentList arg)
  {
    Label status = (Label)getControl("status", Label.class);
    status.setLabel(arg.get("status"));
  }
}
```

Handling a control event on the server

A server-side Java event handler can be registered for each control event. For example:

```
<dmf:button name='button1' onclick='handleClick' />
```

In this example, when the user clicks button1, the WDK framework automatically invokes the event handler, a Java function named handleClick().

Server-side event handlers must have the following signature:

```
public void some_event_handler(Type control, ArgumentList args)
```

where Type is the class or supertype of the control that raised the event.

To define the event handler method on the JSP layout page, use the following syntax. `FormType` is a class or supertype of the `Form` class, and `Type` is the class or supertype of the control that raised the event. You must declare the event handler as a static method and pass the form to it when you define the event handler in a JSP page:

```
<%
    public static void action(FormType
        form, Type control, ArgumentList args)
    {
        ...
    }
%>
```

Updating components with client events

If your component needs to be updated by events on the client, the component class should implement `com.documentum.webcomponent.IClientUpdateEvent`. This interface provides the event handler method `onUpdateData()`. You can fire the `onUpdateData` event from the client when an update of the control is required.

Components that implement `IClientUpdateEvent` include `AbstractInbox`, `VDMView`, `DocList`, and `ObjectGrid`. For example, when a user clicks an object in the object grid component JSP page `objectgrid_classic.jsp`, an `onClickObject` event is fired:

```
<dmf:link name="object_name" datafield="object_name"
    onclick="onClickObject" runatclient="true">
    <dmf:argument name="objectId" datafield="r_object_id">
    </dmf:argument>
    <dmf:argument name="type" datafield="r_object_type">
    </dmf:argument>
</dmf:link>
```

The `onClickObject` client-side event handler (in the same page) fires the server event `onUpdateData`:

```
function onClickObject(obj, id, type)
{
    // update our content component
    postServerEvent(null, null, null, "onUpdateData", "objectId",
        id, "type", type);
    ...
}
```

The `onUpdateData()` event handler method in `ObjectGrid` determines whether the object is a document, a virtual document, or a folder, and performs the appropriate action for each type.

Firing a client event from the server

You can fire a client event from server code using `Form.setClientEvent()`. You may need to fire a client event after some server processing has taken place. When you fire an event from server code, you must register the event handler. Refer to [Registering client event handlers, page 114](#) for information on registering client event handlers.

Do not encode client event arguments using `SafeHTMLString.escapeText()`. Instead, use `escapeScriptLiteral` to encode client event arguments.

The `setClientEvent` method has the following signature:

```
public void setClientEvent(String strClientEventName,  
    ArgumentList clientEventArgs)
```

An example of a client event fired from a control class on the server can be seen with the `GeneralPreferences` class. Its `onCommitChanges()` method sets the user's preferences and then fires a client event to refresh all of the frames except the content frame:

```
ArgumentList args = new ArgumentList();  
args.add("exclude", "content");  
setClientEvent("RefreshFrames", args);
```

The event handler is contained within the same frame's JSP page or JavaScript file:

```
function onRefreshFrames(strExclude)  
{  
    for (var iFrame=0; iFrame < window.frames.length; iFrame++)  
    {  
        window.frames[iFrame].location.reload();  
    }  
}
```

Linking controls by events

You can link controls in several ways:

- Set up conditional value assistance for attributes in Documentum Application Builder

For conditional value assistance, refer to the documentation for Documentum Application Builder.

- Set up a defaultonenter control that fires a button event

You can set the `defaultonenter` attribute on an editable control such as text, filebrowse, or password to true, and then set a button default attribute to true. This will fire the event on the button control when the user hits the **Enter** key on the keyboard while in the editable control.

- Set up an event handler for one control that enables another control

Refer to the following example.

Example 12-34. Activating a dependent control

In the following example, you have a Car Model control (CML) that displays values only when the user has selected a value in the Car Make control (CMK). Initially, you set the dependent control to be disabled (`disabled="true"`). In your initial control, specify a server-side `onselect` event handler:

```
onselect="activateControl"
```

In your component event handler, activate the dependent control:

```
public void activateControl(DropDownList oList, ArgumentList oArgs)
{
    m_list = Integer.parseInt(oList.getValue());
    if (m_list != null)
    {
        DropDownList o2ndList = (DropDownList) getControl("CMK", DropDownList.class);
        o2ndList.setEnabled(true);
    }
}
protected int m_list = 0;
```

State change events

When a form is reloaded, the form requests each control on the form to look for updated state in the HTTP request. If a state update is present, the control overwrites its current state. The form then interrogates each control to determine whether its state has changed and whether the control has a change event handler. If there is a change event handler, the form fires the change event.

The change event handler signature in the server class is:

```
public void event_Name(Type control)
```

The change event handler signature in a JSP page is:

```
public static void event_name(FormType form, Type control)
```

The control `onchange` event is set by `ControlTag.setControlProperties`. For example, `dqlEditor.jsp` contains a dropdownlist control with an `onchange` event:

```
<dmf:dropdownlist name="<%=DQLEditor.CONTROL_MAXRESULTS%/"
    onchange='onSetMaxResults'>
```

The `onchange` event is handled in the DQL editor component class, `DqlEditor`, by a method that has the same name (`onSetMaxResults`):

```
public void onSetMaxResults(DropDownList list)
{...}
```

How control events are raised

Control events are raised when a form URL requests operations on the server. The form (URL) is posted again as a recall operation. The URL that is generated is in the following form (substitute actual values for italicized values):

```
http://1source_UI.jsp?2__dmfRequestId=requestID&3__dmfAction=action&__  
4dmfHandler=handler&5__dmfControl=control
```

1 JSP page

(Required) The URL for the source JSP page. If the URL is not the same as the originating form, the form processor interprets the URL as a jump rather than a recall operation.

2 __dmfRequestId

(Required) Unique ID for the URL. The request ID is automatically generated by the FormTag class into a hidden form field.

3 __dmfAction

(Required) The action is the name of the method to call as the event handler.

4 __dmfHandler

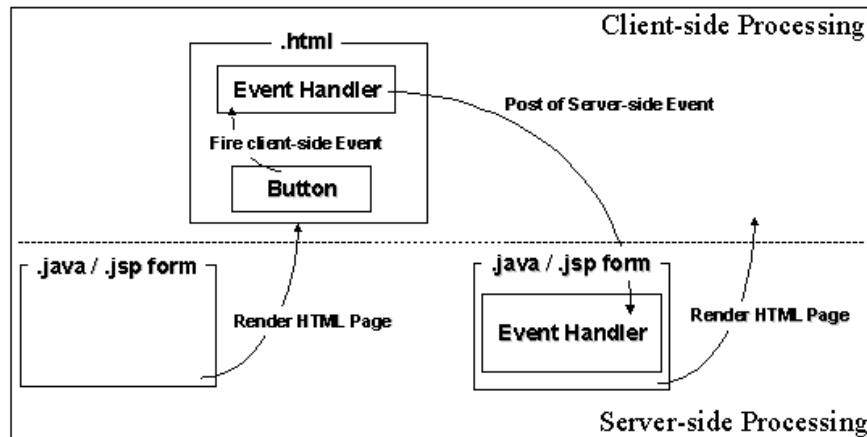
Name of the control that handles the event. If the handler parameter is not set, the top-level form is assumed.

5 __dmfControl

Name of the control that raised the event.

The following diagram illustrates the interaction between client-side and server-side processing:

Figure 12-5. Client-side and server-side event processing



1. The WDK form, which consists of a layout JSP page and a Java behavior class, is processed on the server-side, resulting in HTML being sent to the browser.

2. The user clicks on a button contained within the HTML.
3. The "onclick" event is handled by an event handler, either as client-side JavaScript event handler or a server-side Java event handler. The client-side event handler handles the user's selection through dynamic HTML (one less round trip) or posts the event to server-side code.
4. If the event handler posts a server-side event, a new request is sent back to the J2EE server.
5. The server-side event handler is called by the form processor. The form processor may call business logic, update the state of the form, or navigate to another form.

For example, when the about component is called with the enableTools parameter set to true, the following JSP tag is compiled by the application server:

```
<dmf:button ...onclick='onDQLEditor' runatclient='true' .../>
```

The application server renders the following snippet of HTML to the browser to display a button that launches the dqleditor component:

```
<span title="DQL Editor Button">
<table border=0 cellspacing=0 cellpadding=0 name='About_btnDQL_0'
  onclick='setKeys(event);safeCall(onDQLEditor,this);'...>
<tr style='cursor:hand' height=16>
...
<td style='cursor:hand' background='/wdk525sp1/wdk/theme/kaleidoscope/images/
dialogbutton/bg.gif' nowrap align=center class=buttonLink>DQL Editor</td>
...</tr></table></span>
```

The client side event handler onDqlEditor() raises a server-side event that nests to the dql component:

```
<script>
function onDQLEditor
{
  postComponentNestEvent(null, "dql", "dql");
}
```

Using modal windows

Modal windows provide a performance enhancement in web applications that use several frames. With a modal window, other frames do not need to refresh after the modal frame closes. All non-modal frames are collapsed when a non-modal frame is presented.

Some components that use modal windows are containers, advanced search, login, import file selection, prompt, single and repeating attributes UI pages, and add from clipboard. By default, all nested and contained components are modal and all other component navigation is not modal. Nested components are set to modal by the WDK

framework. If you do not want your nested component to be modal, call `setModal(false)` in your component `OnInit()` method.

Modality hides all frames except the current modal frame and then restores frames on completion of the modal transaction. The modal window can be referenced by an application JavaScript variable `getTopLevelWnd().modalWnd`.

You can explicitly set modality for a form or component in the `OnInit()` event handler. Call the Form method `setModal()` to launch a page in a modal window. The modal window will be displayed within the current frame. The Form class has an `isModal()` method that gets whether the form should be displayed in a modal window. The following example sets modality in a component class:

```
public void OnInit(ArgumentList args)
{
    super.OnInit(args);

    // overwrite the modality
    setModal(false);
    ...
}
```

If your component is in a container and you wish it to be non-modal, add the following lines to the container `OnInit()`:

```
public void OnInit(ArgumentList args)
{
    super.OnInit(args);
    Control control = getContainer();
    if(control instanceof Form)
        ((Form)control).setModal(false);
    setModal(false);
    ...
}
```

To open a modal window outside the application frameset

1. Call `launchModalDialog()`. This JavaScript function is in `/wdk/include/formnavigation.js`.
2. Specify the windows parameters in `launchModalDialog`: URL, window title, window width and height, and window resizeability flag.

To add tracing of modal windows:

1. Open `WebformScripts.properties` in `/WEB-INF/classes/com/documentum/web/form` and add the following line:

```
XX_Modal.trace=true
```

where XX matches the number of the line that includes `modal.js`, for example:

```
09_Modal.href=/wdk/include/modal.js
09_Modal.language=javascript1.2
09_Modal.trace=true
```

- Restart the application server and exercise a component that uses a modal window, such as a properties. A popup window displays trace output as in the following excerpt:

```
modal.js: Resizing Frame : timeoutcontrol
modal.js: Resizing Frame : titlebar
modal.js: Resizing Frame : view
modal.js: Resizing Frame : toolbar
modal.js: Resizing Frame : browser
modal.js: Resizing Frame : workarea
modal.js: Resizing Frame : divider
modal.js: Resizing Frame : menubar
modal.js: Resizing Frame : content
modal.js: Resizing Frame :
modal.js: Resizing Frame :
modal.js: Resizing Frame : status
modal.js: Resizing Frame : messagebar
modal.js: Resizing Frame : statusbar
```

You see the names of all framesets below the top WDK application frame. The two unnamed frames are from nest.jsp.



Caution: The JavaScript file modal.js does not handle frameset that have both rows and cols attributes, although this is a valid HTML construct. A workaround is to rewrite the framesets to use nested framesets, with one frameset having the rows attribute and the other having the cols attribute.

Setting event handlers programmatically

A control can launch an action or operation through a UI button. To set the event handler programmatically, call the control class `getEventNames()` method. To add new event names, override this method in your control class.

Specify the name of the event in the control event handler using the method signature `void eventName()`. The following example sets an event handler for a button:

```
public void onInit(ArgumentList arg)
{
    Button btn = (Button)getControl("the_button", Button.class);
    btn.setEventHandler(Button.EVENT_ONCLICK, "onButtonClick", this);
}
```

The operation event is handled by a method defined in the JSP page that contains the control. In the following example, the onclick event handler is `onOkUserSelect`:

```
<dmf:button ... onclick="onOkUserSelect" label="Ok" ...>
</dmf:button>
```

The `onOkUserSelect` event handler is defined as a method in the component class:

```
public void onOkUserSelect(Control control, ArgumentList args)
```

{...}

Control lifecycle events

A control goes through a lifecycle. At each stage of the lifecycle, the control can be in a different state, have different capabilities, and can raise events to notify your component class. A control has the following lifecycle:

1. **Pre-creation.** A control is pre-created if it has a name and the control is referenced in a form or component `onInit()` method. Some attributes can be set on the pre-created control. Any attribute except the name can be overwritten by a setting in the JSP page when the control is rendered.
2. **Create.** A control is in the create state when it is rendered for the first time. When it is created, the control's `onInit()` event handler is called and the control's internal `isInitialized()` method returns true.
3. **Render.** Every time a form is rendered, some or all of the form's controls are rendered. Controls on an invisible panel are not rendered, but other invisible controls are rendered. The control tests the `isVisible()` method to determine how to render the control. The control rendition method names HTML input elements by calling the helper method `getElementName()`.
4. **Update.** When an event is processed on a form, all form controls are updated. The framework calls `updateStateFromRequest()` on each control, passing the `HttpServletRequest` object as a parameter. The framework calls the control's `hasChanged()` method to determine whether to fire a change event.
5. **Raise an event.** Client-side code can raise a server-side event by calling `postServerEvent()`. This JavaScript function takes a control object as an optional parameter. Alternatively, a control can raise an event by passing itself as the event parameter.

Events maintain the control lifecycle and state, and events launch control operations. Events are modeled as named methods; this means that when a control fires an event, it calls a method with the specified name. All events are synchronous.

Validating a control value

The form processor validates controls on a form (JSP page). Validation is implemented by configurable validation controls. Each validation control checks one input control for a specific type of error condition and displays a message if an error is found.

When a server-side action event is fired on a form, the processor validates the form before calling any event handlers. If a control is not valid, the event is still fired. Your control event handler, in the component that is using the control, must handle the validation error. For example, your component can call `getIsValid()` to ensure that all controls have passed validation.

You can call the Form class method `validate()` to validate controls on a JSP page after application logic has changed input controls.

Example 12-35. Validating controls on a JSP page

The `CheckinContainer` class validates controls in the `onOk()` event handler, when the user submits the checkin form:

```
public void onOk(Control button, ArgumentList args)
{
    validate();
    boolean bValid = getIsValid();
    if (bValid == false)
    {
        return;
    }
    //do checkin logic
}
```

An input control that contains a null or empty value is assumed to be valid by all validators except for the required field validator, which will not accept a null or empty field. Use `requiredfieldvalidator` if your control must have a value.

The `BaseValueValidator` class, which extends `BaseValidator`, is the base class for most validator controls because it returns true for null or empty strings. If your control should not accept null values, extend `BaseValidator` to throw an exception for null or empty values.

Validating a repository object

All validator controls extend the `Label` control and implement the `IValidator` interface. This interface defines three methods: `validate()`, `getIsValid()`, and `getErrorMessage()`.

The base implementation class is `BaseValidator`. This class does the following:

- Accepts the name of the control to validate
- Provides an error message if validation fails
- Overrides `doValidate()`
- Maintains state when events are fired

There are three ways to pass the object ID to your JSP page or component for validation:

- Set the ID as the value of a query attribute in the docbaseobject tag, and then display attributes for the object. For example:

```
<dmfx:docbaseobject name='f1' src="dm_folder where object_name=
'System'"/>
<table><tr><th>Name, Object, Attribute</th></tr>
<tr><td>
  <dmfx:docbaseattribute name='folder' object='f1'
  attribute='object_name' />
</td></tr>
</table>
```

- Set the ID as the value of a Java expression. For example:
- ```
<dmfx:docbaseobject name='f1' id='<%= (String)pageContext
.getAttribute("idF1") %>' />
```
- Set the ID in your component class by calling DocbaseObject.setObjectId(). You should also make object ID a required parameter for your component class.

#### Example 12-36. Passing an object ID for validation

The following example from DeleteDocument initializes a DocbaseObject control with the object ID:

```
public void onInit(ArgumentList arg)
{
 super.onInit(arg);
 m_strObjectId = arg.get("objectId");
 m_strFolderId = arg.get("folderId");
 ...
 // Initialise the Docbase Object
 DocbaseObject docbaseObj = (DocbaseObject) getControl(
 "object", DocbaseObject.class);
 docbaseObj.setObjectId(m_strObjectId);
 ...
}
```

## Adding a control listener

You can add control listeners to allow other classes to be notified of control lifecycle events.

The Prompt class adds a control listener in its onInit() method:

```
addControlListener(this);
```

The listener class provides the IControlListener implementation to handle the onControlInitialized event, which is fired by every control when it is initialized.

#### Example 12-37. Implementing a control listener

In the following example from the Web Publisher class ReadOnlyListener, the onControlInitialized() sets a DocbaseAttributeValue control to read-only:

```

if (bReadOnly)
{
 class ReadOnlyListener implements IControlListener
 {
 public void onControlInitialized(Form form, Control control)
 {
 // set Docbase attribute value controls as read-only
 if (control instanceof DocbaseAttributeValue)
 {
 DocbaseAttributeValue value = (DocbaseAttributeValue)control;
 value.setReadOnly(true);
 }
 }
 }
};

// Add listener
addControlListener(new ReadOnlyListener());
}

```

## Creating custom pseudoattributes

To add a pseudoattribute type for a repository type, open the docbaseobjectconfiguration definition for the repository type. For example, the default configuration for the display of `dm_sysobjects` is found in `/webcomponent/config/library/docbaseobjectconfiguration_dm_sysobject.xml`. A pseudoattribute type `rich_text` is defined. The definition specifies tag classes to handle the display and saving of the `rich_text` attribute:

```

<scope type="dm_sysobject">
<docbaseobjectconfiguration id="attributes">
<names>...</names>
<types>
 <attribute type="rich_text" repeatingonly="false" singleonly="false">
 <tagclass>com.documentum.web.formext.control.docbase.
 RichTextDocbaseAttributeTag
 </tagclass>
 <labeltagclass>com.documentum.web.formext.control.docbase.
 RichTextDocbaseAttributeLabelTag
 </labeltagclass>
 <valuetagclass>com.documentum.web.formext.control.docbase.
 RichTextDocbaseAttributeValueTag
 </valuetagclass>
 <!-- optional elements -->
 </attribute>
</types>
...

```

For more information about configuring custom formatters, handlers, tag classes, or configuration elements for a pseudoattribute, refer to [Modifying the display and handling of attributes, page 395](#).

To configure a pseudoattribute type for an attribute list, add it to the `attributelist` definition for the appropriate scope. For example, the `rich_text`

type is used to define a pseudoattribute for folder descriptions in the attributes\_dm\_folder\_docbaseattributelist.xml file:

```
<scope type="dm_folder">
<attributelist id="attributes" extends="
 attributes:webcomponent/config/library/attributes_docbaseattributelist.xml">
1<pseudo_attributes>
 2<attribute name="folder_description" 3type="rich_text" category="info"
 display_after="title">
 <label_text>Description</label_text>
 </attribute>
</pseudo_attributes>
...
```

The pseudoattribute is then used in a JSP page. To use the same example, the folder description is displayed with a rich text control in the newfolder.jsp page:

```
<!-- Rich text description. Use the panel to prevent display if Rich Text is
not installed -->
<dmfx:richtextpanel>
 <tr>
 <td scope="row" width="10%" align="right">
 <dmf:label nlsid="MSG_DESCRIPTION_COLON"/>
 </td>
 <td width="90%" align="left" colspan="3">
 <dmfx:richtexteditor name='<%=DocList.FOLDER_DESCRIPTION%>' hasImages='true'/>
 </td>
 </tr>
</dmfx:richtextpanel>
```

The richtexteditor control name DocList.FOLDER\_DESCRIPTION maps to the folder\_description pseudoattribute.

- 1** Contains pseudoattributes that are attached to the object type. The pseudoattribute must be handled by a custom tag class.
- 2** Name of an attribute that does not exist in the repository. The attribute is handled by the classes that are specified for the attribute in the docbaseobjectconfiguration definition. For more information, refer to [Modifying the display and handling of attributes, page 395](#).
- 3** Type of the pseudoattribute as specified in the docbaseobjectconfiguration definition. For more information, refer to [Modifying the display and handling of attributes, page 395](#).

To add a pseudoattribute programmatically, call DocbaseObject.addPseudoDocbaseAttribute().

## How controls and tags work together

Controls and tags interact as follows:

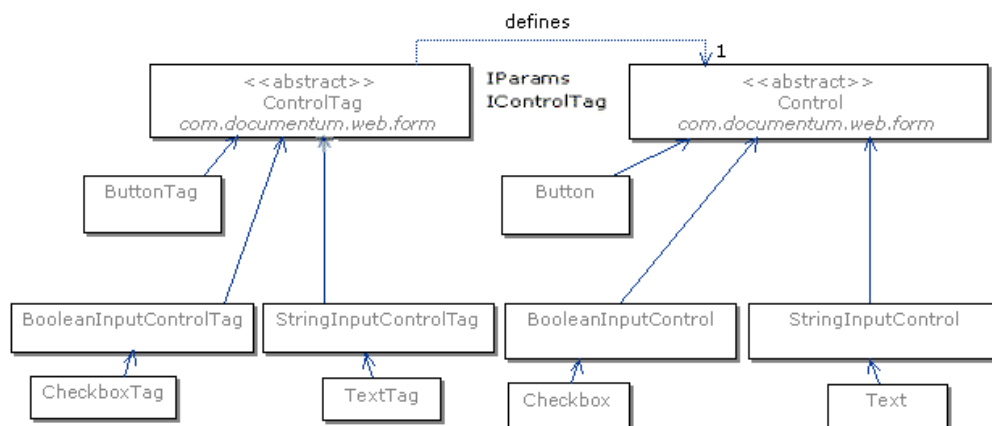


1. The Web designer sets default values for the control attributes on the JSP page.
2. The JSP page is requested by the client browser
3. The UI control display is initialized using the control attribute values set on the JSP page and rendered to the browser as HTML and JavaScript.
4. A user changes a control value.
5. The new value is submitted when the form is submitted, along with any other control changes.
6. The form then updates the control state on the server and also performs any operation that is triggered by form submission.

An instance of a Control subclass lasts for the lifetime of the control, while an instance of a ControlTag subclass lasts only within the lifetime of the HTTP request.

The following figure diagrams the relationship between the Control and ControlTag classes.

**Figure 12-6. Control and ControlTag relationship**



## Control arguments

Controls can use the ArgumentTag class to centralize the use of control constants and to pass event arguments to an event handler. The class provides methods to access the attributes of the parent tag. The argument tag should always be contained within another tag.

Both the basic controls and the repository-enabled controls have an ArgumentTag class. The formext.control.ArgumentTag class extends form.control.ArgumentTag and adds a contextvalue attribute to the tag.

**Example 12-38. Passing arguments to a component or action**

The following example sets the values for the `actionmultiselectcheckbox` tag arguments using the `argument` tag.

```
<dmfx:actionmultiselectcheckbox name='check' value='false'>
 <dmfx:argument name='objectId' datafield='r_object_id'/>
 <dmfx:argument name='type' datafield='r_object_type'/>
 <dmfx:argument name='lockOwner' datafield='r_lock_owner'/>
 <dmfx:argument name='folderId' contextvalue='objectId'/>
</dmfx:actionmultiselectcheckbox>
```

Argument tags are required for every action that can be applied to the selected objects as well as for the components that are launched by those actions. The above example passes all of the arguments to the selected action, but one action may use only some of the arguments and another action may use different arguments.

The arguments are passed to the action class that launches the multiple action. The action class passes the arguments on the component. For example, the delete component gets the arguments of the object ID and its containing folder as follows:

```
public void onInit(ArgumentList arg)
{
 super.onInit(arg);
 m_strObjectId = arg.get("objectId");
 m_strFolderId = arg.get("folderId");
 ...
}
```

Additional utility classes and methods can be found in `com.documentum.web.form.Util` and the `com.documentum.web.util` package.

# Customizing Components

The Component class and derived classes contain the logic for business component behavior. All components that use WDK 5 functionality must extend Component. The base Component class includes component lifecycle event handlers.

You can manage and translate messages and labels for your component using the NLS service (refer to [Using messages and labels, page 236](#)). Components can be included within other components (refer to [Including a component in another component, page 240](#)), and you can give a group of components a common UI through the use of a container (refer to [Configuring containers, page 243](#)).

Component APIs are described in the following topics:

- [Component base class, page 436](#)
- [Component public interface, page 436](#)
- [Navigating within and between components, page 437](#)
- [Implementing failover support, page 442](#)
- [Implementing a component, page 445](#)
- [Using a component listener, page 447](#)
- [Accessing an included component, page 449](#)
- [Supporting drag and drop, page 450](#)
- [Customizing containers, page 457](#)
- [Multi-repository support, page 463](#)
- [Component dispatching, page 467](#)
- [Component lifecycle, page 469](#)
- [JSP page processing \(form processor\), page 470](#)
- [Form classes, page 475](#)

## Component base class

The Component class extends the Form class and provides the following component-specific support:

- Implementation of the parameters defined in the definition XML file
- Helpers for navigating between components and within components
- Helpers for retrieving component-specific configuration values
- Helpers for accessing the repository
- Call-backs for container support

If your component does not need to support behavior in addition to that provided by `com.documentum.web.formext.component.Component`, you can simply use the Component class for your custom component class in the component definition. In this case, your component definition can have the following static settings: JSP start page, scope qualifier, `nlsbundle`, and `helpcontextid`.

## Component public interface

Each component supports a public interface through which all other components and containers communicate. The component interface consists of the following elements:

- Parameters

Parameters initialize the component. Parameters are configured in the component definition.

- Events

Events are raised by controls in the component JSP pages and handled in the component behavior class.

- Properties

Properties get or set the component state. Properties can be set in the user interface or programmatically by the component behavior class. Default property values can be supplied by custom elements in the component configuration file.

A component can be extended to add support new functionality, without changing the component's caller. Components inherit or override the contractual behavior of the component that they extend.

# Navigating within and between components

The following topics describe navigation from one component to another and within a component:

[Navigating within a component, page 437](#)

[Jumping to a component, page 437](#)

[Nesting to another component, page 438](#)

[Returning to the calling component, page 440](#)

[Returning to a component, then jumping to another, page 440](#)

[Navigating within a container, page 441](#)

For information on container navigation, refer to [Customizing containers, page 457](#).

## Navigating within a component

Use the method `setComponentPage()` in your event handler to jump to a named page in a component.

### Example 13-1. Changing the component JSP page

In the following example, the import container definition specifies a page named `importupload`. jumps to the `importupload` page as named in the container configuration file:

```
<pages>
 ...
 <importupload>
 /webcomponent/library/importContent/importUpload.jsp
 </importupload>
</pages>
```

The `onOk()` event handler of the container class sets the file path, format, content type, import folder ID, file name, category, and descendants information, and then navigates to the `importupload` page:

```
setComponentPage("importupload");
```

## Jumping to a component

The method `setComponentJump()` jumps to another component. You can call `setComponentJump()` from within an event handler. The form processor will add the `JUMP` argument to the redirect URL and forward to the target URL. The state of the calling component is lost when you jump to another component.

**Example 13-2. Jumping to another component**

The parameters for `jump` and `nest` are as follows:

- String name of component (required)
- String component start page (optional)
- Argument List (optional)
- Context

In the following example from `Subscriptions`, the `navigateToFolder()` method calls `setComponentJump()` in order to jump to the `drilldown` component and adds a folder ID argument for the folder to jump to:

```
ArgumentList args = new ArgumentList();
args.add("folderId", strFolderId);
setComponentJump("drilldown", args, getContext());
```

Your component can also jump to a component in the `browsertree` component using the `Component` class method `jumpToTargetComponent(ArgumentList args)`. The argument list that is passed to this method should contain a `nodeIds` parameter, which is the name of the node in the tree to jump to. The method `jumpToTargetComponent` then calls `setComponentJump()`.

**Example 13-3. Jumping to a node in the browser tree**

In this example from the `administration` component, the `onInit()` function calls `jumpToTargetComponent` to jump to the requested node in the tree:

```
{
 jumpToTargetComponent(args);
 super.onInit(args);
 ...
}
```

## Nesting to another component

Server-side nested navigation displays one or more JSP pages, maintaining the state of the calling JSP page. The state of the calling component is maintained on a nested call and on return.

Nest to another component by calling `setComponentNested()`. The method `setComponentNested()` takes the same set of parameters as `setComponentJump()` with the addition of an `IReturnListener` argument. The return listener is called when the nested component returns, and return results and arguments can be passed back to the listener. For more information about the return listener, refer to [Using a component listener, page 447](#).

You can added arguments to the nested form with the method `addFormNestedArgs(ArgumentList arg)`.

**Example 13-4. Passing arguments to a nested component**

You can call the `nest` method in a server-side event handler in a Java class or in a JSP page scriptlet. The following example shows how you pass arguments from the argument list to a nested component. First assemble your arguments to pass to the nested class:

```
args.add(Prompt.ARG_DONTSHOWAGAIN, "true");
setComponentNested("prompt", args, getContext(), this);
```

In your nested component, you get the arguments and use them. For example:

```
String strDontShowAgain = args.get(ARG_DONTSHOWAGAIN);
```

**Example 13-5. Passing arguments from a nested component**

You can pass arguments from the nested component back to the calling component using the `Form` class method `setReturnValue()`. You should call your nested component using the action service, either with a user-initiated action through an `actionlink` or `actionbutton` or with a call to the action service in your component class. The following example from the `reportmainlist` component is an event handler for the link `Edit Settings`. The event handler calls the nested component `reportmainsettings`.

```
public void onEditSettings (Control control, ArgumentList args)
{
 ...
 args.add(ReportMainSettings.PARAM_OVERDUE_DAYS, String.valueOf(m_overdueDays));
 ActionService.execute("reportmainsettings", args, getContext(), this, this);
}
```

In the nested component `reportmainsettings`, the user selects settings that are passed to the calling component.:

```
public boolean onCommitChanges ()
{
 // validate
 Text textCtrl = (Text) getControl(OVERDUE_DAYS_TEXT_CONTROL_NAME, Text.class);
 m_overdueDays = Integer.parseInt(textCtrl.getValue());
 ...
 setReturnValue(PARAM_OVERDUE_DAYS, new Integer(m_overdueDays));
 return true;
}
```

In the calling component class `ReportMain`, the `onComplete()` method is called when the action completes, and this method gets the returned arguments:

```
public void onComplete (String strAction, boolean bSuccess, Map completionArgs)
{
 if (strAction.equals("reportmainsettings") && bSuccess)
 onReturnFromEditSettings(this, completionArgs);
}
```

The arguments are used in the method `onReturnFromEditSettings()`:

```
Object obj = map.get(ReportMainSettings.PARAM_FILTER);
m_overdueDays = ((Integer) map.get(
 ReportMainSettings.PARAM_OVERDUE_DAYS)).intValue();
```

## Returning to the calling component

Server-side return navigation returns to the calling page, which can be the same page. The method `setComponentReturn()` takes no parameters, so state is dropped for the component. You can use this call in an event handler for a cancel button or to return to the same page after some other user event has been processed.

To return to the caller from a contained component, use the following call:

```
((Component) getTopForm()).setComponentReturn();
```

**Note:** Do not call `setComponentReturn()` in an `onExit()` event handler, because the component form has been unloaded from memory, and the form dispatcher is unable to return to the caller.

### Example 13-6. Return to a calling component in an event handler

In the following example from the About component, the `onClose()` button event handler returns to the calling page:

```
public void onClose(Button control, ArgumentList args)
{
 setComponentReturn();
}
```

**Note:** If you are calling `setComponentReturn()` from an included component, such as a component within a container, you must call `setComponentReturn()` on the parent component. For example, the advanced search component class handles the `onCloseSearch` event as follows:

```
public void onCloseSearch(Control control, ArgumentList args)
{
 Form topform = getTopForm();
 if (topform instanceof Component)
 {
 ((Component) topform).setComponentReturn();
 }
 else
 {
 topform.setFormReturn();
 }
}
```

Alternatively, if you are jumping from a contained component to another component outside the container, call `setComponentReturnJump` to exit the modal contained component.

## Returning to a component, then jumping to another

An additional method, `setComponentReturnJump()`, returns to the calling page and performs a jump to another component. If you are jumping from a contained



component, which is by nature modal, to a component outside the container, you must call `setComponentReturnJump()`, which returns to the container and then to the outside component.

**Example 13-7. Return and jump to another component**

In the following example, the advanced search component jumps to the search component in its `doSearch()` method:

```
setComponentReturnJump("search", args, context);
```

## Navigating within a container

You can define any order of page presentation for the **Next** and **Previous** buttons in a container. You can implement the `onNextPage()` method either in the contained component or in the container class, depending on your business case.

**Example 13-8. Navigating to the next or previous component**

In the following example, the `finishworkflowtask` component class `onNextPage()` method determines the current page and navigates appropriately depending on the current page:

```
public boolean onNextPage()
{
 boolean retValue;
 String page = getComponentPage();

 // we're on the assign performers page
 if(page.equals("assignperformers"))
 {
 setComponentPage("finish");
 retValue = true;
 }

 // we're on the Finish page
 else
 {
 retValue = false;
 }
 if(retValue == true)
 {
 updateControls();
 }
 return retValue;
}
```

In this example, the code first gets the current component page (`getComponentPage()`) in order to navigate to the appropriate next page. If the current page is `assignperformers` (named `<pages><assignperformers>` in the `finishworkflowtask` component definition),

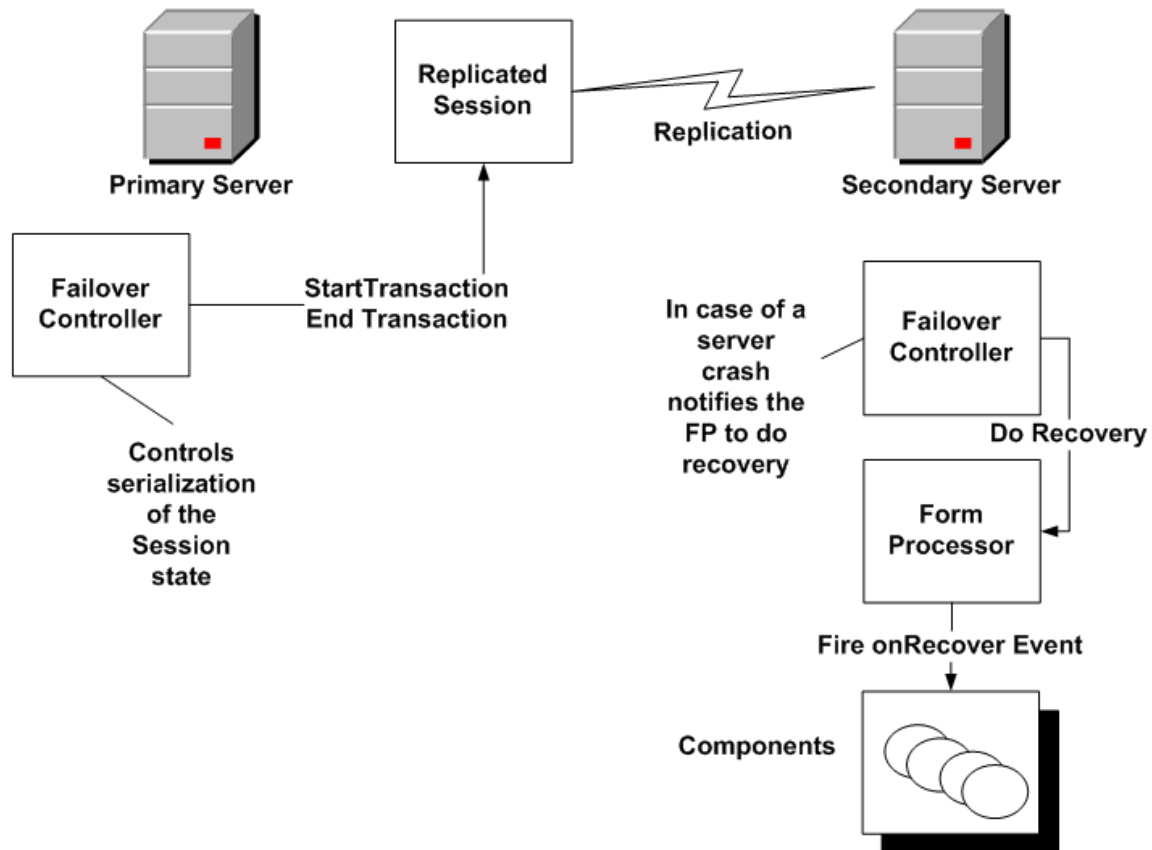
this method navigates to the finish page and updates the controls.. If the current page is the finish page, then `onNextPage()` does nothing.

## Implementing failover support

The topic [Configuring application failover support, page 89](#) describes how to enable failover support for an application and its existing components. This topic describes how to implement failover support in a custom component.

Failover is controlled by the servlet `WDKController`, which intercepts each request and serializes the state. The failover servlet sets a flag before state is serialized and removes it after state has been serialized. When the controller encounters a request with the flag still set, it detects a failover state and instructs the `FormProcessor` to call `onRecover()` on all components in memory. This process is diagrammed below.

Figure 13-1. Serialization process



The Control class implements serializable, so all components inherit this implementation. Perform the following additional steps to support failover in your component:

1. Define the component as failover-enabled by adding the element `<failoverenabled>true</failoverenabled>` to the component definition

If the component is included within a container, verify that the container definition supports failover. If not, the component will not be serialized.

2. Verify that failover is enabled in the application configuration file (`/custom/app.xml`) and in the application server. (Consult the application server documentation for information on enabling failover or session serialization as well as the WDK client installation guide for WDK-specific configuration for supported clustered application servers.)
3. Decide which variables within your component should not be serialized. Hide data that should not be serialized by placing the keyword `transient` before the data type in the variable declaration, for example:

```
private transient Class m_pageClass = null;
```

Use the following criteria to decide whether a variable should be transient:

- Mark as serializable or transient all member variables of Control and Component subclasses. (You can also mark variables as Externalizable, which allows a custom implementation of Serializable.) Example:

```
private transient Class m_pageClass = null;
```

- Make sure that there are no static variables that store session state. Java does not serialize static variables. Use the class `SessionState` to store session state. Example from `AcIValidate`:

```
SessionState.setAttribute("accessorlist", accessorNames.toString());
```

- Mark member DFC objects as transient. DFC objects are not serializable and should not be stored in the HTTP session beyond the request scope. For DFC objects that must be serialized, instantiate the object using methods of `IDfClientX`, mark it externalizable, and use the `serialize` and `deserialize` methods of `IDfClientX` to persist the object. BOF classes are also not serializable

**Note:** Some WDK components that are not yet serialized may instantiate DFC objects, which would prevent your extending the component and rendering it serializable.

- Check for large objects, such as objects created for caching purposes, and sensitive information such as passwords, to mark as transient.
4. Override `onRecover()` to do any necessary cleanup and recovery after the application server restores the session:
    - Initialize variables that are marked as transient; if you do not, these variables will have unexpected null values after recovery.

- Call `super.onRecover()` before your cleanup and recovery code. For example, if you have a custom checkin component, you could implement `onRecover()` to check whether there any unfinished file uploads, remove the file, and retry the upload.

For actions, implement recovery in the action `execute()` method. Make sure that no state (instance variables) is associated with action and precondition classes. Their instances are used as singletons, and state caching would cause concurrency issues.

5. Check for non-serializable data using the `FAILOVER` tracing flag (see below).

**Testing for non-serializable data** — All session attributes in WDK classes are either serializable or transient. To find session attributes that are not serializable, use the `FAILOVER` flag in tracing. Non-serializable attributes will produce a runtime exception with a message that a non-serializable attribute has been placed in the session. The following type of tracing statement will be generated in the log (default location `DOCUMENTUM_HOME/logs/wdk.log`):

```
188193 [http-8080-Processor25] DEBUG com.documentum.web.common.Trace -
Object in session is not serializable: preferredrenditionsservice,
Reason: com.documentum.web.formext.config.ConfigService$LookupFilter
```

You can turn on the `FAILOVER_DIAGNOSTICS` tracing flag. WDK will print the total size of the session (in bytes that are serialized at the end of each request). This is a very expensive call and should be used only for debugging purposes.

You can also test for non-serializable data by using Tomcat 5, which persists session data and restores it by default. Tomcat writes serialization errors to the console and log. (You must add a section to the `server.xml` configuration file to turn off this feature.) Other application servers have configuration parameters that turn on failover support. Consult the application server documentation for details.

The `WDKController` servlet filter intercepts requests from the application server. The filter sets a flag to keep track of the request and detect failover. In case of a failover, the load balancing mechanism routes the request to the next available server in the cluster. The request is intercepted by the filter, which detects the failover and marks the session to indicate that is a recovered session. All forms are notified to perform recovery. Restored components will perform cleanup and resume operations.

The following example shows how the advanced search component persists the query and reexecutes it in `onRecover()`.

#### **Example 13-9. Implementing `onRecover()`**

Search queries are persisted. The `advsearch` class `AdvSearchEx` delegates recovery in `onRecover()` to `SearchInfo` and then re-executes the query:

```
public void onRecover()
{
 super.onRecover();
 if (m_searchInfo != null)
```

```

 {
 m_searchInfo.onRecover();

 // re-execute
 if (authenticateSources() == true)
 {
 executeQuery();
 }
 }
}

```

Note that the call to `Form::onRecover()` is a simple check to see whether recovery is needed.

The `SearchInfo` implementation persists the query definition as follows:

```

private String m_strQueryDef=null;

public void setSmartListDefinition(
 IDfSmartListDefinition idfSmartListDefinition)
{
 .../
 m_idfSmartListDefinition = idfSmartListDefinition;
 m_strQueryDef = null;
 if (m_idfSmartListDefinition != null)
 {
 IDfQueryDefinition querydef = m_idfSmartListDefinition.
 getQueryDefinition();
 ...
 }
}

```

`SearchInfo` then recovers the query in `onRecover()`:

```

public void onRecover()
{
 m_idfSmartListDefinition = null;
 if (m_strQueryDef != null)
 {
 ...
 this.setSmartListDefinition(SearchUtil.getSmartListDefinition(
 m_strQueryDef));
 setInstanceId();
 ...
 }
}

```

## Implementing a component

All content components should provide similar functionality and behavior and well as similar layout. Your component must implement the following lifecycle methods and functionality:

**onInit()** — Should accept appropriate arguments and supply valid defaults for non-mandatory parameters. Set up controls to valid initial defaults. Set connections on

databound controls. (Refer to [Getting data in the component class, page 391](#) for an example.) Read standard and custom component definition settings such as object type filter values, object types to display, and visible columns.

**Example 13-10. Implementing Component.onInit()**

Your form or component class should call the `onInit()` method of the super class:

```
public void onInit(ArgumentList args)
{
 super.onInit(args);
 ...
}
```

**onRender()** — Read and apply standard and custom preferences such as the size of data lists to display. Read preferences during every `onRender()`, as they may have been modified by the user between invocations of the component. Set the data provider before the page is rendered.

**Example 13-11. Setting the data provider in onRender()**

If your component needs to get data, you can get a data provider in the `onRender()` method. Get the session in your component `onInit()` function. The following example from `AclWhereUsed` sets a data provider:

```
public void onRender()
{
 super.onRender();
 Datagrid datagrid = ((Datagrid)getControl(CONTROL_GRID, Datagrid.class));
 datagrid.getDataProvider().setDfSession(getDfSession());
}
```

You can also initialize other controls in the `onInit()` function, such as a `datadropdownlist` or `label`, read preferences, or perform other component functions that do not require user input.

**Example 13-12. Supporting columns of Attributes**

If your component displays a list of docbase objects with various attributes shown in columns, the component should support the configuration of columns of attributes for display.

In the following example from the `Web Publisher` component class `ChannelList`, the `readConfig()` method is called from the component `onInit()` lifecycle event handler. The method `readConfig()` reads the columns definition and processes it:

```
protected void readConfig ()
{
 // read the list of visible column
 m_columns = new ComponentColumnDescriptorList(this, "columns");

 // hide locale column if global content management turned off.
 if (getGlobalContentManagementFlag() == false)
 {
 m_columns.remove(LANGUAGE_CODE_COL);
 }
}
```

```

 }
 ...
}

```

**Data refresh** — Override the Component API method "onRefreshData()". The framework calls this method after the execution of an action that has invalidated the data being displayed by this component. Call "refresh()" on all Data controls to automatically re-query the data or rebuild custom ScrollableResultSet objects as required. For an example, refer to [Refreshing data, page 394](#).

**Object type filters** — If your component displays a list of Documentum objects, the component should provide a dropdown control that allows users to change the type of objects displayed, for example, Folders, files, all objects, or some custom object type (the latter requires a custom filter).

**Clipboard operations** — The standard cut, copy and paste clipboard operations should be supported if appropriate. The component should implement the clipboard operation handler interfaces IClipboardCutHandler and IClipboardPasteHandler. Refer to [Clipboard APIs, page 596](#) for more information.

**Title** — All content components should provide a title to indicate the component name, for example, Inbox. Additionally, content components that display Documentum objects should display the current repository and username in the format "MyComponent (docbase : username)".

**Context** — Update the current component Context if your component allows navigation through the repository or displays a repository location. For example, if the component is viewing the contents of a folder, then it could set the current folderId in the context.

## Using a component listener

IReturnListener can be used to perform operations after a nested component returns to the calling component. The return listener is called when the nested component returns, and return results and arguments can be passed back to the listener. You can add arguments to the nested form with the method addFormNestedArgs(ArgumentList arg).

### Example 13-13. Using a listener to force a refresh

In the following example from ObjectGrid, which implements IReturnListener, the onReturn() implementation forces a refresh on the grid when navigation returns to the grid:

```

public void onReturn(Form form, Map map)
{
 // force refresh on dataproviders
}

```

```

 m_datagrid.getDataProvider().refresh();
}

```

To perform some business logic after your component returns from a nested component, provide your listener class as a parameter to the Container class method `setComponentNested()`.

#### Example 13-14. Listening to a nested component

In the following example, the `Permissions` class declares a listener in the `onChangeAcl()` method call to `setComponentNested()`. The listener gets selections from a locator in the nested component:

```

public void onChangeAcl(Link link, ArgumentList args)
{
 ArgumentList selectArgs = new ArgumentList();
 selectArgs.add("component", "acobjectlocator");
 selectArgs.add("flatlist", "true");
 setComponentNested(
 "acobjectlocatorcontainer", selectArgs, getContext(),
 new IReturnListener()
 {
 public void onReturn(Form form, Map map)
 {
 // Handle return event from select permission set component.
 if (map != null)
 {
 LocatorItemResultSet setLocatorSelections = (
 LocatorItemResultSet)map.get(ILocator.LOCATORSELECTIONS);
 if(setLocatorSelections != null && setLocatorSelections.first() ==
 true)
 {
 String strAclId = (String) setLocatorSelections.getObject(
 "r_object_id");
 if (strAclId != null && !strAclId.equals(m_strSelectedAclId))
 {
 updateControls(strAclId);
 }
 }
 }
 }
 });
}

```

#### Example 13-15. Returning values to a listener

You can return values from a nested component to the caller's listener. The listener has a single method, `onReturn()`, which is called when the component returns. Call `Form.setReturnValue` to pass a `Map` of values to the return listener. The `onReturn(Form form, Map map)` method has a `form` parameter that specifies the form or component that is returning and a `map` parameter that specifies a `Map` of values. In the following example from `ChangePassword`, the `onChangePassword()` method passes several arguments back to the caller:

```

// pass the login credentials back to the caller

```



```

setReturnValue("docbase", strDocbase);
setReturnValue("username", strUsername);
setReturnValue("password", strNewPassword);
setReturnValue("domain", strDomain);
setComponentReturn();

```

The Login class implements IReturnListener, and its onReturn() method gets the values passed from the nested changepassword component:

```

public void onReturn(Form form, Map map)
{
 // authenticate with the new password
 boolean bSuccess = false;
 if (map != null && map.isEmpty() == false)
 {
 try
 {
 String strDocbase = (String)map.get("docbase");
 String strUsername = (String)map.get("username");
 String strPassword = (String)map.get("password");
 String strDomain = (String)map.get("domain");
 if (strDocbase != null || strUsername != null || strPassword != null)
 {
 authenticate(strDocbase, strUsername, strPassword, strDomain);
 }
 bSuccess = true;
 }
 ...
 }
}

```

## Accessing an included component

You can include a component in another component using a <dmfx:componentinclude> tag in a JSP page. This allows you to reuse components within multiple components. The componentinclude tag has a component attribute that specifies the name of the component to be included.

You can access and set values in the included component by getting the componentinclude control in your component class.

### Example 13-16. Accessing an included component

In the following example from ReportDetailsContainer, the container JSP page reportdetailscontainer.jsp includes the reportdetailsheader component as follows:

```

<dmfx:componentinclude component='reportdetailsheader' name=
 '<%=ReportDetailsContainer.HEADER_CONTROL_NAME%>' />

```

The container class accesses the contained componentinclude control as follows:

```

public static final String HEADER_CONTROL_NAME = "__HEADER_CONTROL_NAME";
public void saveReportHeading (StringBuffer csvContent)
{
 ...
}

```

```

// Save the rest of the header info.
ReportDetailsHeader headerComponent = (ReportDetailsHeader) getControl(
 HEADER_CONTROL_NAME, Component.class);
headerComponent.saveReportHeading(csvContent);
}

```

### Example 13-17. Accessing controls in an included component

After you have obtained a reference to the included component, you can get and set values on the controls in the included component. In the following example, the test component includes a changepassword component. The component is included in the JSP page for the test component as follows:

```

<dmfx:componentinclude component='changepassword' name=
 'changepwd' />

```

This code example gets the value of the new password from the named Password control (controls must be named to be accessed in server-side code):

```

Component subComp = (ChangePassword) getControl(
 "changepwd", ChangePassword.class);

String strNewPassword = ((Password) subComp.getControl("newpassword")).getValue();

```

## Supporting drag and drop

The following drag and drop support for the Internet Explorer browser can be globally enabled or disabled in /wdk/app.xml:

- Drag and drop between windows and frames in the same user session of IE

Supported by JavaScript, this feature can be globally enabled or disabled in /wdk/app.xml. Users cannot turn on drag and drop if it is globally disabled in app.xml.

- Additional support for drag and drop to and from the desktop

Supported by an Active-X plugin that can be globally enabled or disabled in /wdk/app.xml

- Initial user preference for using the Active-X plugin

If the initial state is set to false (<initial\_user\_state>), the user must enable the plugin. User is prompted to do this at Set to true to turn on the plugin out of the box.

To disable drag and drop, copy the <dragdrop> element from /wdk/app.xml to /custom/app.xml and set the value of <enabled> to false. To disable the drag and drop plugin, copy the <plugins> element to your custom app.xml file and set the value of <enabled> to false. (This will also disable the rich text editor spellchecker that is included in the plugin.)

Refer to [<dragdrop> element, page 79](#) and [<plugins> element, page 80](#) for more information on these global drag and drop settings.

Drag and drop is not supported for the following conditions:

- Accessibility mode is turned on by user
- Window is created by launching the browser again and getting a new WDK application session. (You can drag and drop with a WDK window that is created with the WDK **New Window** menu item.)
- XML documents dragged to or from desktop
- Virtual documents dragged to or from desktop
- OLE compound documents

For performance reasons, minimal tests are done to ensure that a user may actually execute a drag or drop action. As a result, the user may be presented with a valid drop cursor with one or more actions listed as available although the action will fail when the user attempts the action.

The following topics describe drag and drop customization:

- [Drag and drop support in WDK components, page 451](#)
- [Adding drag and drop to a component definition, page 452](#)
- [Adding drag and drop to a JSP page, page 454](#)
- [Adding drag and drop support to a control, page 455](#)
- [Troubleshooting drag and drop, page 456](#)

## Drag and drop support in WDK components

Drag and drop is implemented in a component by adding a `<dragdrop>` element in your component definition and adding supporting JSP tags in the UI.

The following components support drag and drop:

**Table 13-1. Components that support drag and drop**

Component	Source	Targets
Cabinets, Home Cabinet	Items	Folders, virtual documents, background
Room	Items	Folders, virtual documents, background
Search	Items except external results	Folders, virtual documents

Component	Source	Targets
Subscriptions	Items	Folders, virtual documents, background
Versions, Locations	Items	Folders, virtual documents
Clipboard	Items	Folders, virtual documents
Browser tree	Items	Folders, virtual documents

## Adding drag and drop to a component definition

Any component that inherits from `AbstractNavigation` or from another component that supports drag and drop can enable drag and drop in the component XML configuration file. Components that don't inherit from `AbstractNavigation` can make use of the helper class `DragDropSupportBuilder` in order to support drag and drop in the component definition.

To add drag and drop capability to a component that inherits drag and drop support, add a `<dragdrop>` element to the component definition. This element contains the following configuration settings:

**Table 13-2. Configuration elements (<dragdrop>)**

Element Name	Description
<code>&lt;sourceactions&gt;</code>	Contains zero or more <code>&lt;sourceaction&gt;</code> classes that support drag actions on sources in the component. This element must be declared even if it contains no <code>&lt;sourceaction&gt;</code> elements, in order to enable the component as a drag source.
<code>&lt;sourceaction&gt;</code>	Contains a fully qualified class name of class that implements <code>IDragSourceAction</code>
<code>&lt;targetactions&gt;</code>	Contains zero or more <code>&lt;targetaction&gt;</code> classes that support drop actions on drop targets in the component

Element Name	Description
<targetaction>	Contains a fully qualified class name of class that implements IDragTargetAction, for example, com.documentum.web.formext.control.dragdrop.CopyToFolderTargetAction. To remove a drop action, remove this element from definition.
<dataproviders>	Contains one or more <dataprovider> elements. Remove this element and its contents to prevent the component from being a usable drag source.
<dataprovider>	Contains a <format> element and a <provider> element.
<dataprovider>. <format>	Fully qualified class name for class that provides data for the format. Built-in formats that implement IDragDropData (in com.documentum.web.formext.control.dragdrop): ObjectIdData, ChildIdData, FileDescriptorData , VdmNodeData, FileContentsData, FileDropData
<dataprovider>. <provider>	Fully qualified class name for class that implements IDragDropDataProvider and provides data for the associated format

To support drag and drop from your component to the desktop, your component definition must include the format FileDescriptorData, which provides the filename, size, and modification date of the file to the desktop. Content is then streamed to the desktop. To support drag and drop from the desktop to your component, the target must provide the following:

- The component or control needs an IDropTarget implementation
- The UI control must specify an ondrop handler that is implemented in the control or component class
- At least one action must be available for the UI control.

DocbaseFolderTree and derived controls will support drag and drop if the controls are located on a component whose XML configuration includes a <dragdrop> element.

## Adding drag and drop to a JSP page

To support drag and/or drop in a component page, you must add a `dmfx:dragdrop` control to the page. This control will generate the JavaScript support for drag and drop. You can then surround a drag or drop region on the page with a `dmf:dragdropregion` control. This control will add the HTML markup (approximately 1200 HTML character for each source) that enables the enclosed elements to be dragged, dropped, or both. To enable the enclosed element to be dragged as a source, set the `dragenabled` attribute to `true`.

Inline elements such as icons or labels can be enabled as drop targets.

To enable the enclosed element to be a drop target, set a value for the `dragdropregion` `ondrop` attribute. You can use the drop target default implementation by setting the `ondrop` attribute value to `"onDrop"`. Set at least one value for the `enableddroppositions` attribute.

The `datafield` attribute should match that of the enclosed tag, in order to display a tooltip.

**Dragging** — Most objects that a user can add to the clipboard are potential drag sources. In Webtop classic view, these objects are listed with checkboxes. In streamline view, they are the objects that can be added to the clipboard with the **Add to Clipboard** action link. Examples of sources include objects in a datagrid or object grid; objects in the navigation tree except repositories, cabinets, or built-in nodes such as Inbox and Subscriptions; search results; and objects in specialized views such as versions, relationships, subscriptions, and renditions. The following objects cannot be dragged: cabinets, inbox objects, and administration objects. If the user drags an object that has versions or renditions, the default rendition or current version is used as source unless another specific version or rendition is selected.

**Dropping** — Potential drop targets include locations to which an object in the clipboard can be moved or pasted. Examples of drop targets include virtual documents (in the default rendition) and folders or folder subtypes such as cabinets or rooms. The following objects cannot serve as targets: non-container objects such as documents, windows or frames in a different session, repository node in the navigation tree, inbox, my files, categories, administration, and the root cabinet.

Drop actions supported by the component should be listed in the `<dragdrop>`. `<targetactions>` element of the component definition. The following target action classes are available in WDK, in the package `com.documentum.web.formext.control.dragdrop`.

**Table 13-3. WDK target actions**

Action class	Action displayed
CopyToFolderTargetAction	Copy
MoveToFolderTargetAction	Move
LinkToFolderTargetAction	Link
AddVirtualDocumentNodeTargetAction	Add
RepositionVirtualDocumentNodeTargetAction	Reposition
SubscribeAction	Subscribe
ImportTargetAction	Import (accepts files from desktop)

The `dragdropregion` tag gets the source formats and target actions by calling `IDragDropDataProvider::getFormats` and `IDropTarget::getDropTargetActions`. Your component can implement the formats and actions interfaces to do any necessary business logic.

**Performance** — Each `dragdropregion` tag adds approximately 1200 HTML character for each source. The page load loops through image overlays for each inline `dragdropregion` and positions them over the region tag. Expanding a tree node merges additional image overlays into the tree frame.

You can turn off drag and drop in `app.xml` to compare the page rendering performance.

## Adding drag and drop support to a control

You can integrate drag and drop into a control by the interfaces in the `com.documentum.web.dragdrop` package: `IDragSource`, `IDropTarget`, `IDragSourceAction`, `IDropTargetAction`, and `IDragDropDataProvider`. Controls that integrate drag and drop identify themselves as a drag source, a drop target, or both. These controls are interoperable with other controls that support drag and drop.

The source and target are represented as interfaces `IDragSource` and `IDropTarget`, respectively. Each `IDragSource` and `IDropTarget` supports a set of programmer-defined formats (Class objects) along with a set of programmer-defined drag drop actions. Because the same drag drop motion may correspond to more than one drag drop action, a menu of actions is presented to the user. The list of available source actions is created by adding an `IDragSourceAction` instance for each action to the `IDragSource` implementation. The list of available target actions is created by adding an `IDropTargetAction` instance for each action to the `IDropTarget` implementation. If a

source action is specified with the same name as the target action invoked, then it will be run after successful target action invocation.

A component can override the event handler for all of the component's contained controls by implementing the Form class event handler `onDrop()`.

## Troubleshooting drag and drop

You can look at the following sources for an error in which an object cannot be dragged or dropped:

1. Uncomment the lines in the `.Odnd` style in `/wdk/theme/documentum/css/dragdrop.css` and refresh the page. If an item is enabled for drag and drop, it will be highlighted in purple. If it is not highlighted, check the JSP page to make sure there is a `dmf:dragdropregion` tag and one or more `dmf:dragdropregion` tags on the page.
2. If the object is enabled but does not accept a dragged item, view the source in the browser. Look for the JavaScript `initDragDrop` with drop actions such as `Move Here`, as in the following example

```
<script type='text/javascript'>
 initDragDrop('HomeCabinetClassicView_0', '
1114551362046', 'HomeCabinetClassicView_0', '
HomeCabinetClassicView_DragDropRegion_0', '
objectId~0b000001803097ff|parentObjectId~0c000001801de8bc',
true, true, 'com.documentum.web.formext.control.dragdrop.
ObjectIdData,com.documentum.web.formext.control.dragdrop.
FileContentsData,com.documentum.web.formext.control.dragdrop.
ChildIdData,com.documentum.web.formext.control.dragdrop.
FileDescriptorData', 'Move here\tmove\t\tover\t\tcom.
documentum.web.formext.control.dragdrop.ChildIdData\ttrue\
nImport\timport\t\tover\t\tcom.documentum.web.formext.control.
dragdrop.FileDropData\ttrue\nLink here\tlink\t\tover\t\tcom.
documentum.web.formext.control.dragdrop.ObjectIdData\tfalse\
nCopy here\tcopy\t\tover\t\tcom.documentum.web.formext.
control.dragdrop.ObjectIdData\tfalse', 'onDrop');
</script>
```

3. If you do not see actions listed, check the component configuration file to see whether the component is enabled for drag and drop and that target actions are defined. The above example from the home cabinet classic view inherits drag and drop configuration from the `homecabinet_list` component, with the following target actions defined (class name shortened for display purposes):

```
<targetactions>
 <targetaction>
 com.documentum.web...dragdrop.CopyToFolderTargetAction
 </targetaction>
 <targetaction>
 com.documentum.webweb...dragdrop.MoveToFolderTargetAction
 </targetaction>
```



```
<targetaction>
 com.documentum.web...dragdrop.LinkToFolderTargetAction
</targetaction>
<targetaction>
 com.documentum.web.web...dragdropImportTargetAction
</targetaction>
</targetactions>
```

## Customizing containers

The Container class extends the Component class and provides additional container-specific support for components within the container:

- Notifiers are called on the contained components to determine whether changes can be committed or canceled when the user selects **OK**, **Cancel**, or **Close**. (Refer to [Implementing container notifications, page 458.](#))
- Helpers are called on the contained components to determine when the next and previous buttons should be shown and enabled. The helpers are called when the user selects **Next** or **Previous** to determine which page to display. (Refer to [Accessing components within containers, page 460.](#))
- You can call a container programmatically. (Refer to [Passing arguments in a container, page 462.](#))

For information on calling a container by an action or URL, refer to [Calling containers, page 247.](#)

## Calling a container from a server class

Contained components can be invoked from server-side code by component navigation methods. You can invoke a contained component through the `Component.setComponentJump()` and `setComponentNested()` methods. These methods take the following arguments:

- `strComponentName`: The component to jump to
- `strStartPage`: The component Start Page (optional)
- `arg`: The component arguments (optional)
- `context`: The context within which to call the component (refer to [Context, page 487](#) for more information on context)
- `returnListener`: An implementation of `IReturnListener`, with `setComponentNested()` only. Refer to [Using a component listener, page 447](#) for more information.

**Example 13-18. Jumping to a container**

In the following example from NodeManagement, the `onClickBreadcrumb()` method jumps to the administration container:

```
public void onClickBreadcrumb(Breadcrumb breadcrumb, ArgumentList args)
{
 Breadcrumb breadCrumbControl = (Breadcrumb)getControl(
 CONTROL_BREADCRUMB, Breadcrumb.class);
 String strPath = breadCrumbControl.getValue();

 int index = strPath.lastIndexOf('/');
 if(index != -1)
 {
 String componentNLSName = strPath.substring(index+1);
 if(componentNLSName.equals(getString("MSG_ADMINISTRATION_LINK")))
 {
 setComponentJump("administration",getContext());
 }
 }
}
```

## Implementing container notifications

Components that update data can use the change notification methods of the container class. The query will inform the container whether the contained component's changes can be committed, cancelled, or reverted and report the changes that are being committed, cancelled, or reversed. The following change notification methods are available:

**canCommitChanges()** — Called by the container to determine whether changes can be committed. Returns true unless you override this implementation to invoke your business logic. The OK button on the dialog and wizard containers is disabled if this method returns false.

**Example 13-19. Testing whether component can commit changes**

In the following example, the Checkin class overrides `canCommitChanges()` to determine whether to proceed with checkin:

```
public boolean canCommitChanges()
{
 if (getDoNotCheckin() == true)
 {
 return true;
 }
 else
 {
 return ((m_strCheckoutPath != null) &&
 (m_strCheckOutPath.length() > 0));
 }
}
```

**onCommitChanges()** — Called by the container when changes are to be committed (when the user selects **OK**).

**Example 13-20. Committing component changes**

In the following example from `AdminDelete`, the `onCommitChanges()` method performs the commit:

```
public boolean onCommitChanges ()
{
 destroyObject(m_objectId);
 return true;
}
```

**boolean canCancelChanges()** — Called by the container to determine whether changes can be cancelled. In the Component class, this method returns true. The Cancel button on the dialog and wizard containers is disabled if this method returns false.

**Note:** If both `canCommitChanges()` and `canCancelChanges()` return false, a Close button is displayed in place of the OK and Cancel buttons.

**Example 13-21. Setting component cancel changes conditions**

You can override `canCancelChanges()`, which returns true for the Component class. In the `JobStatus` class, the `canCancelChanges()` implementation returns false, because the component is a viewer, not an editor:

```
public boolean canCancelChanges ()
{
 return false;
}
```

**onCancelChanges()** — Called by the container when changes are to be cancelled, that is, when the user selects **Cancel** or **Close**. Returns whether the changes were successfully canceled. In the Component class, this method returns true.

**Example 13-22. Implementing onCancelChanges()**

In the following example from `NewCabinet`, the **Cancel** button has the following event handler (`onCancelChanges()` is called by the container's parent `onCancel()` event handler):

```
public boolean onCancelChanges ()
{
 if (m_strNewObjectId != null)
 {
 deleteCabinet(m_strNewObjectId, getDfSession(), false);
 m_strNewObjectId = null;
 }
 return true;
}
```

**requiresVisitForCommit()** — This method is available in the `propertysheetcontainer` class. It is called by the container on an uninitialized component to determine whether the component must be committed for its container to commit. This method looks up a

requires `visitForCommit` configuration attribute value in the container definition. Refer to [Require visit, page 249](#) for more information.

**`canRevertChanges()`, `onRevertChanges()`** — These methods return true in the Component class. You should implement `onRevertChanges()` to return false if changes cannot be reverted.

## Accessing components within containers

Switch between components in the container in your container class by calling `setCurrentComponent()` or `getContainedComponent()`, generally in an event handler method. Contained components are not initialized in the container's `onInit()` method unless you explicitly initialize them. You can initialize all your contained components using `setCurrentComponent()`.

Call `getContainedComponent()` to get the current component in a container. If you need to access a non-current component or multiple components in the container, you can iterate through the array that is returned by `getContainedComponents`.

### Example 13-23. Initializing contained components

You can initialize all of the contained components using `setCurrentComponent()` and `initContainedComponent()`. The following example from `TaskMgrContainer` does initializes the contained component and returns to the previously selected component:

```
protected void initAllVisibleComponents(Tabbar tabs)
{
 String currentCompId = getContainedComponent().getComponentId();

 // iterate through the tabs and initialize the components
 for (Iterator iter = tabs.getTabs(); iter.hasNext() == true;)
 {
 Tab tab = (Tab)iter.next();
 if (tab != null && tab.isVisible() == true)
 {
 // set it as current to allow initialization
 setCurrentComponent(tab.getName());
 initContainedComponent();
 }
 }

 // restore the original current component
 setCurrentComponent(currentCompId);
}
```

### Example 13-24. Getting the current component

If you need to get a single component, call `Container.getContainedComponent()`. In the following example from `LocatorContainer`, `getContainedComponent()` returns the current component:

```

public void onInit(ArgumentList args)
{
 // remember initial selection
 String strSelectedIds = args.get("selectedobjectids");
 args.remove("selectedobjectids");

 super.onInit(args);

 // transfer initial selections to the current component
 Component compCur = getContainedComponent();
 if (compCur != null && compCur instanceof ILocator)
 {
 ((ILocator) compCur).setSelections(strSelectedIds);
 }
}

```

### Example 13-25. Accessing specific components in a container

Use `getContainedComponents()` to access all of the components in the container. In the following example from `SaveReportLocator`, a specific component is accessed:

```

ArrayList componentList = getContainedComponents();
int count = componentList.size();
for (int index = 0; index < count; index++)
{
 Object obj = componentList.get(index);
 if (obj instanceof SysObjectLocator)
 {
 //do what needs to be done
 }
}

```

**requiresVisit attribute** — The `requiresVisit` attribute on the component element in a container definition requires the component to be visited before an OK button is displayed. Refer to [Require visit, page 249](#) for more information.

The Container class provides the following methods that support page navigation in wizards:

**hasPrevPage()** — Called by the container to determine whether there is a previous page. The Previous button on wizard containers is disabled if this method returns false.

**Note:** The Next and Previous buttons are hidden if both `hasNextPage()` and `hasPrevPage()` return false.

### Example 13-26. Testing whether the container has a previous or next Page

In the following example from the Web Publisher `publish` component class, the `hasPreviousPage()` and `hasNextPage()` methods are called by the container class to determine whether to display **Previous** or **Next** buttons. The call to `getComponentPage()` returns the current page. The page "selectcabinets" is named in the publish component definition as `<pages>.<selectcabinets>`.

```

public boolean hasPrevPage()
{

```

```
String thisPage = getComponentPage();
if (thisPage.equals("selectcabinets"))
 return true;
else
 return false;
}
```

**onNextPage()** — Called by the container when the user selects **Next**. The method returns true if the page was successfully switched.

**onPrevPage()** — Called by the container when the user selects **Previous**. The method returns true if the page was successfully switched.

**hasNextPage()** — Called by the container to determine whether there is a next page. The Next button on wizard containers is disabled if this method returns false. Multi-page components override `hasNextPage()` and `hasPrevPage()` to add paging, typically using `setComponentPage()`.

## Passing arguments in a container

The Container class adds the contained component arguments to its own arguments. Control values are propagated within the container only if the control implements the `getValue()` and `setValue()` methods. Documentum attribute controls do not implement these methods and do not propagate changed values within the container.

Some of the most commonly used methods of the Container class to get or set contained components or their attributes are described below:

**get/setContainedComponentArgs()** — The method `setComponentArgs()` passes arguments from the container to contained components. The method `getContainedComponentArgs()` gets arguments from the container, within the contained component.

### Example 13-27. Passing arguments from a container to components

The following example from `NewFolderContainer` gets the arguments from the contained components and replaces them with other arguments:

```
private void updateComponentArgs()
{
 // set up arguments with type and objectid of new object
 ArgumentList args = getContainedComponentArgs();
 ArrayList components = getContainedComponents();
 NewFolder component = (NewFolder)components.get(0);
 String strNewObjectId = component.getNewObjectId();
 String strType = component.getNewType();
 args.replace("objectId", strNewObjectId);
 args.replace("type", strType);
 setContainedComponentArgs(args);
}
```

```

Context context = getContext();
context.set("objectId", strNewObjectId);
context.set("type", strType);
}

```

### Example 13-28. Getting container arguments in the contained component

The following example from `AclComponent` gets the container instance and gets its arguments by calling `getContainedComponentArgs()`:

```

Form topForm = getTopForm();
if(topForm instanceof Container)
{
 Container topContainer = (Container) topForm;
 String objectId = topContainer.getContainedComponentArgs().get("objectId");
 if(objectId != null && !objectId.equals("") && !objectId.equals("newobject"))
 {
 //Do business logic
 }
}
}

```

## Multi-repository support

The following multi-repository support is provided in WDK:

- Objects can be copied or linked across repositories

Copy creates a new object that is a copy (replica or mirror object) of the selected version of the source object. Deep folder copy is supported. Link creates a reference object (shortcut). Move is not supported.

- Content transfer actions on replica and reference objects can be performed
- Inbox and workflow can include objects in other repositories

Only the current repository is queried. Users can select attachments from multiple repositories to attach to a workflow. These distributed attachments are treated as foreign objects.

- Search can operate on multiple repositories
- My Files lists the user's recently used replica objects and checked out replica, reference, or foreign objects in other repositories.

Only the current repository is queried for My Files. When the user checks out an object in another repository, a reference object is created in the user's home cabinet.

For information on creating federations and managing replication, refer to *Distributed Configuration Guide* for Content Server.

**Note:** Subscriptions are not supported on reference, or foreign objects. You must log in to the source repository in order to subscribe to an object.

To see multi-repository objects in the inbox or workflow, the remote repositories must configure a `dm_DistOperations` job. Refer to *Distributed Configuration Guide* for details.

The following topics describe multi-repository support:

- [Replica \(mirror\), reference, and foreign Objects, page 464](#)
- [Adding multi-repository support to a component, page 465](#)
- [Scoping and preconditioning actions on remote objects, page 466](#)
- [Session management with multiple repositories, page 466](#)

## Replica (mirror), reference, and foreign Objects

The WDK application will attempt to get a session in the source repository using the current username and credentials in order to perform actions on replica, reference, or foreign objects. The action will fail if the user credentials are not the same for the current and source repositories.

A replica object is a mirror of the object in the source repository. Replication objects are created by a replication job on the Content Server. Replica objects are displayed in drilldown and list views of objects, and write operations on replica objects can be performed on the source object. Apply lifecycle on a replica object is not supported.

A reference object consists of a shortcut to an object in another repository. The reference object mirrors the attributes of the remote object. Reference objects can be created by the user with Paste as Link action across repositories. The Content Server also creates reference objects for distributed checkout, distributed workflow, and distributed virtual documents.

A foreign object is an object ID that is the same as a the object ID of an object in a remote repository. Foreign objects are available in distributed workflows and multi-repository search:

- Workflow tasks do not perform a query, so that attributes on the foreign object are not accessible.
- Search queries the object in the remote repository for text or attributes that meet the search criteria. When the user performs an operation on a foreign object in search results, the operation is performed after login to the repository in which the object is located.

Many actions can be performed on reference objects and on foreign objects, for example: checkin, checkout, cancel checkout, edit, view properties ( local properties), comment, and find target action (jump to the source to perform other actions). To determined whether an action on a reference or foreign object is supported, check the lists of unsupported actions in `mirror_undefined_actions` and `foreign_undefined_actions`. For more information, refer to [Scoping and preconditioning actions on remote objects, page 466](#).



## Adding multi-repository support to a component

Objects from multiple repositories will be exposed in your custom components if you display the contents of a cabinet or folder, objects that have been modified by a user, or inbox/workflows. In the JSP page, include a `<dmf:argument>` tag for `i_is_replica` and `i_is_reference` so that icons will display properly for all reference or replica objects. For example, the `relationships_classic.jsp` page adds these arguments to the action multiselect checkbox:

```
<dmfx:actionmultiselectcheckbox name="check" value="false">
 <dmf:argument name="objectId" datafield="r_object_id"/>
 ...
 <dmf:argument name="isReference" datafield="i_is_reference"/>
 <dmf:argument name="isReplica" datafield="i_is_replica"/>
</dmfx:actionmultiselectcheckbox>
```

These attributes must also be added to the query. In the same example class, `Relationships`, the attributes are added to the query parameter:

```
private static final String INTERNAL_ATTRS = "
 sysobj.r_object_id,...i_is_reference,i_is_replica ";
```

If you are adding an icon that represents the object type, add the `isreplicadatafield` and `isreferencedatafield` attributes to the control, similar to the following from `relationships_classic.jsp`:

```
<dmfx:docbaseicon ...isreplicadatafield='i_is_replica'
 isreferencedatafield='i_is_reference' size='16' />
```

If your component needs to perform an operation on the object in the remote repository, such as running a query, add the `<setrepositoryfromobjectid>` element to `true`. By default, all actions in the context of the component are performed on the local replica object, reference object, or foreign object ID. This element should be set on the container rather than on the component, if your component exists within a container.

**Note:** Containers and components must have the same repository session.

The following utility classes can provide information to your component:

- `DocbaseUtils::isForeign(String strObjectId)`  
Returns true if the object ID is a foreign object
- `DocbaseUtils::isReference(String strObjectId)`  
Returns true if the object ID is a reference object
- `DocbaseUtils::getDocbaseNameFromId(IDfId objectId)`  
Returns the repository name in which the source object exists

## Scoping and preconditioning actions on remote objects

Two pseudo-types have been created to support scoping of actions or components for mirror or foreign objects: `mirror_dm_sysobject` and `foreign_dm_sysobject`. The `DocbaseTypeQualifier` method `getParentScope()` returns the `r_object_type` for the source of the mirror or foreign object.

The WDK configuration files that scope actions by the remote object pseudotypes are in `/wbcomponent/config/actions`: `mirror_undefined_actions` and `foreign_undefined_actions`. You can add actions to these files to prevent an action from operating on a replica or foreign object. You cannot remove an action from these files unless you create a custom action that effects the action in the remote repository.

You can add preconditions to an action that allow the action to execute only if the object is a reference or foreign object. `RemoteObjectPrecondition::queryExecute` returns true if the object is a replica or foreign object. `ReplicaObjectPrecondition::queryExecute` returns true if the object is a replica.

## Session management with multiple repositories

DMCL manages multiple connections for a single session. The user logs in once, and then the username and password are saved and used for remote connections. The username and password must be the same for the remote repository when the user attempts an action on a replica, reference, or foreign object.

Some actions on foreign IDs, replicas, or references are directed to the source object: checkout, checkin, and cancel checkout. Remote queries or transactions are not performed except in search, which queries all selected sources.

To query a remote repository, don't use `Component::getDfSession()`. Instead, use the following:

```
IDfSessionManager sessionManager = SessionManagerHttpBinding.
 getSessionManager();
IDfSession session = null;
try
{
 session = sessionManager.getSession(remote_repository);
 // code to construct query against foreign repository
 ...
 query.execute(session, IDfQuery.DF_CACHE_QUERY);
 ...
}
catch
{
 ErrorMessageService.getService().setNonFatalError(...);
}
finally
```

```
{
 if (session != null)
 sessionManager.release(session);
}
```

## Component dispatching

The component dispatcher servlet allows a container to call or include a component. The dispatcher maps the component URL to the appropriate implementation URL and dispatches the component. The dispatch process is described in [How components are dispatched, page 467](#).

## Component dispatcher servlet

The component dispatcher maps the component URL to the appropriate page URL. The component dispatcher is implemented as a Java servlet. The servlet is registered in the J2EE web server web.xml file, so that URLs that begin "*root\_context/component*" are routed to the dispatcher:

```
<servlet>
 <servlet-name>ComponentDispatcher</servlet-name>
 <servlet-class>com.documentum...Component.ComponentDispatcher
</servlet-class>
</servlet>

<servlet-mapping>
 <servlet-name>ComponentDispatcher</servlet-name>
 <url-pattern>/component/*</url-pattern>
</servlet-mapping>
```

## How components are dispatched

The component dispatcher performs five steps:

1. Resolves the component name

The dispatcher extracts the component name from the URL. The dispatcher also retrieves the start page name from the URL, if specified.

2. Creates a dispatch context

The context is a list of name/value pairs that are recognized by the configuration service. The dispatcher creates the context based on component parameters

and configuration service scope qualifiers. Each qualifier registered with the configuration service maps a scope name to one or more context names. When the dispatcher finds a match between a component parameter name and a qualifier context name, the parameter value is used to construct the dispatch context.

The dispatcher does not test all URL arguments. It tests only the arguments whose parameter names are defined in the component definition.

3. Finds the component definition

The dispatcher converts component parameters in the dispatch context into scope values and then compares these values with the scopes that are specified in the in-memory component definitions. For example, the `deletedocument` component definition has a scope of type and required parameter names `objectId` and `folderId`. The configuration service checks the registered context names and finds that the `DocbaseTypeQualifier` class has registered `OBJECTID` as a context name for the type qualifier (scope). The dispatch bridge creates a context that contains the name/value pair `objectId/aaaabbbbccccdddd` and forwards this context to the component class.

4. Creates the dispatch bridge

Each page implementation type has a designated dispatch bridge to handle that type of page. The component dispatcher will use the behavior class specified in the component configuration file to determine which dispatch bridge must be called. The dispatch bridge ensures that context, state, or other component behavior is set up before forwarding to the start page.

Two dispatch bridges are provided in the WDK framework:

- WDK 5 component dispatch bridge: Dispatches WDK 5 pages.
- Default dispatch bridge: Dispatches HTML or raw JSP pages.

5. Dispatches the component

## WDK 5 component bridge

The WDK 5 component bridge calls components that contain WDK JSP pages. The dispatch bridge sets the dispatch context and determines the appropriate component definition whose scope matches the context. The bridge navigates to the start page for the scoped component. The WDK Form processor takes over and processes the JSP page.

When components are scoped, the component behavior class is determined at run time. The dispatcher maps the URL context to configuration service qualifiers, and the scope that most closely matches the context is used to look up the scoped component definition. The behavior class and NLS bundle are read from the scoped component definition and added to the request before it calls the start page.

## URL bridge (default)

The default bridge is called when the component behavior class element is missing in the component definition. This bridge will forward to the URL specified as the start page element value of the component definition. The URL may point to an HTML, ASP, or JSP page. The features of the Component class will not be available to the page that is dispatched.

## Component lifecycle

Components are dispatched by the component dispatcher, which evaluates the user context and returns the appropriate component definition. The Form processor processes the component JSP pages and generates lifecycle events.

The processor handles multiple requests of the same page or component in the following way:

- The first request initializes and renders without validation
- Subsequent requests fire change events, action events, and re-rendering
- The final request fires change events and form exit

Component instances are destroyed by the WDK history mechanism: The last ten components are held on a stack to support the browser back button. When the next component is launched, the last one drops off the stack and is garbage-collected. All component instances are destroyed when the Web session times out. Refer to [Browser history, page 98](#) for more information.

The lifecycle methods are called by the form processor in the following order:

- `onInit()`  
Called when the JSP page or component is first requested.
- `onRender()`  
Called every time a form is requested by URL, after the event handlers are called but before the JSP processing takes place. Do not call a navigation method after `onRender()` has been called within a request, because the response may already have been written.
- `onRefreshData()`: Called when the form data is changed. Your component should override `onRefreshData()`, call `super()`, and then add code to refresh datagrids or other controls.
- `onRenderEnd()`: Fired for every request, after all JSP form processing has completed (after the `</dmf:form>` tag). Ensures that resources are cleaned up. For example, if you acquire a pooled connection in `onRender()`, you need to release it in `onRenderEnd()`.

- `onExit()`  
Called when the application navigates to another form or component.

**Note:** A lifecycle event handler method on a JSP page takes precedence over a handler method in a server-side class.

## JSP page processing (form processor)

Component JSP pages are processed and validated by the form processor. The following topics describe the functions of the form processor:

- [What the form processor does, page 470](#)
- [Form processing sequence, page 471](#)
- [Processing browser navigation, page 473](#)
- [Form navigation operations, page 473](#)

For information on configuring form processor properties, refer to [Navigation defaults, page 97](#).

For information on control validation, which is performed by the form processor, refer to [Validating a control value, page 428](#).

## What the form processor does

The form processor interprets HTTP requests and translates them into WDK method calls and events. The form processor performs the following functions:

- Fires events during a form's lifecycle:  
  
Change events are fired by controls that accept user input. When a form is reloaded, it requests each control to look for updated state in the HTTP request. If it finds updated state, the control overwrites its state. The form then interrogates each control for state changes and for a defined change event handler. If there is a change event handler, the form fires the change event and passes to the handler the control that changed. The handler has access to the old and new values.  
  
Action events are fired when a form reloads. When a form is reloaded, the HTTP request will specify any action event handler that is specified for controls on the form, along with the ID of the control that will handle the event and, optionally, the ID of the control that raised the event. The form processor fires the action event.
- Performs validation of controls within the form
- Maintains a configurable history trail that models the history in the browser (refer to [Navigating using browser history, page 241](#))

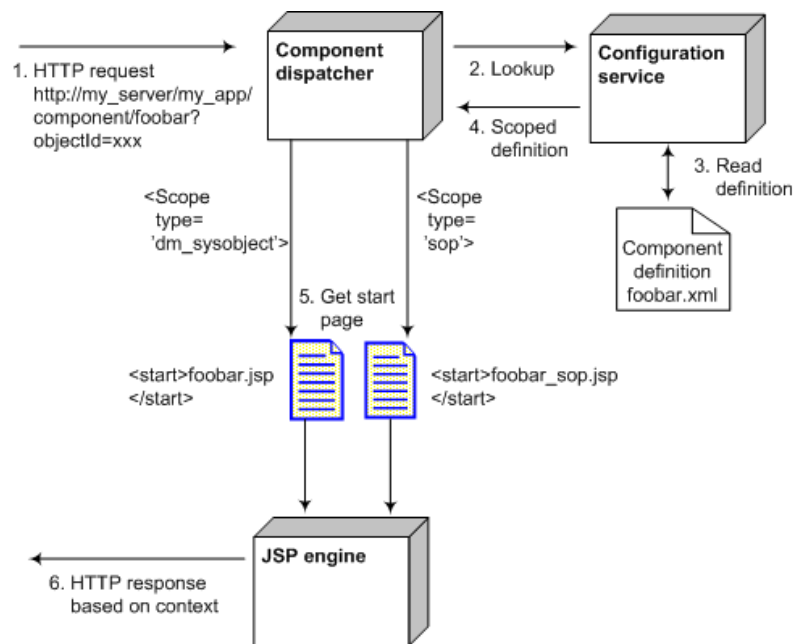
- Registers form processor hooks
- Registers lifecycle listeners.
- The form processor executes the form navigation operations. Most of the operations request a jump to another form or page in the operation execute() method.

If a jump is requested, the processor performs a JSP include to the requested page and closes browser history around nested forms. If no jump is requested, the processor performs a JSP include to the operation's redirect page.

## Form processing sequence

The sequence of processing that occurs when a component is called via URL is diagrammed below.

**Figure 13-2. Component Processing Sequence Diagram**



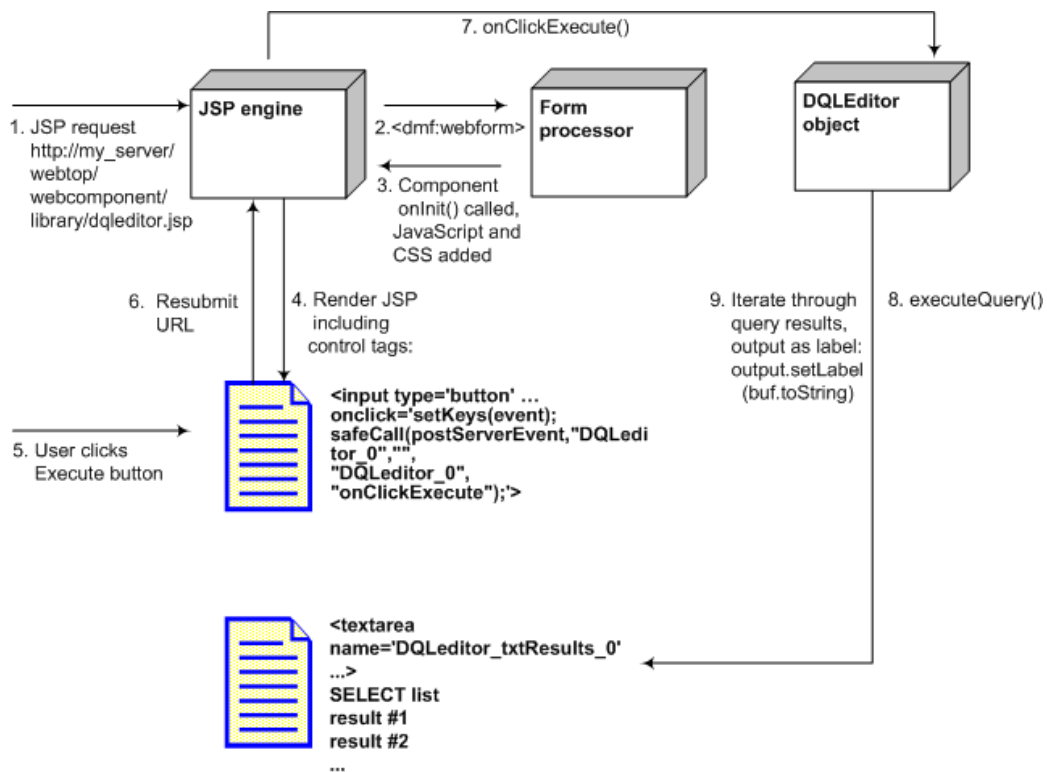
In the above diagram, a typical component is processed in the following sequence:

1. An HTTP request goes to the Web server. The component is called by URL, which must include parameters that are defined as required in the component definition. The URL can also contain optional parameters.
2. The component dispatcher calls the configuration service to look up the component definition.

3. The configuration service finds the component that is named in the URL and reads its definition.
4. The configuration service returns the component definition for the scoped HTTP request. In the example, the context is provided in the object ID. The configuration service matches this context to the closest component definition. In the example, the match is to the object type.
5. The component dispatcher dispatches the component start page for the context. The implementation can be either a WDK 5 component or a URL.
6. The JSP engine renders the HTTP response (in HTML and JavaScript). This last step is composed of many processes, which are outline in the following paragraphs.

The interaction between a component JSP page, user, and UI controls is diagrammed below:

**Figure 13-3. JSP page, control, and user interaction diagram**



In the above diagram and example, user entries in a JSP page are processed in the following sequence:

1. The start page for the DQL editor component is processed by the JSP engine.



2. The `<dmf:webform>` tag class tells the form processor that the JSP page is a WDK 5 page and should be processed by the form processor. If the webform tag arguments for behavior class and NLS properties are not provided in the JSP page, the form processor looks for them in the context. The component dispatcher sets these in the context from the component definition.
3. The form processor calls `onInit()` on the component class and adds JavaScript and CSS includes to the output. (Components that are included in a container using the `<dmf:containerinclude>` tag are initialized when the containerinclude tag is processed, unless the container initializes its contained components in its `onInit()` method.)
4. The JSP engine renders the HTML output. The output for one control, the **Execute** button, is shown in the diagram.
5. The user clicks **Execute** in the UI.
6. The URL is resubmitted, and the `onClickExecute` server event is posted.
7. The `onClickExecute()` event handler method in `DQLEditor` is called. Among other functions, this method executes the query that the user has entered in the UI.
8. The `DQLEditor` object executes the query using an `IDfQuery` object (`execute()` method). The query results are returned as an `IDfCollection` object.
9. The `DQLEditor` object iterates through the `IDfCollection` of query results and formats them as a `Label` object. The editor writes the results to the UI using `Label.setLabel()`.

## Processing browser navigation

The default behavior when the users selects the browser **Back** button is that the form processor returns to the URL (component or Web page) from which the user's current component was called.

If a user hits the browser button in a component JSP page that was not called from within the application, the form processor will find no return page. The form processor and form return operation redirect to a no return page. The no return page displayed can be configured in the `FormProcessorProp` properties file by setting the `noReturnURL` entry. The default page is `/wdk/blank.htm`.

## Form navigation operations

The Form operations are executed by the form processor in response to navigation requests in the URL. The table below describes the types of form navigation operations. Each operation is represented by an operation class with the same name, in the package

com.documentum.web.form. The form operations classes extend FormOperation, the base class.

**Table 13-4. Form navigation operations**

Operation	Description
HistoryReleasedOperation()	Handles a request for a history snapshot that has been released and requests a forward to /wdk/historyReleased.jsp
IncludeOperation()	Handles a request for an included form.
JumpOperation()	When the URL contains a JUMP_TYPE parameter with a value of JUMP, the operation requests a forward to the target form and closes the old form
PageJumpOperation()	When the URL contains a JUMP_TYPE parameter with a value of PAGE, the operation requests a forward to the specified page in the specified target form and closes the old form.
NestOperation()	When the URL contains a JUMP_TYPE parameter with a value of NESTED, the operation requests a forward to the specified page in the specified target form. The old form remains open.
RecallOperation()	When the URL does not contain an action argument, the recall operation requests the processor to call the current form again.
ReturnOperation()	When the URL contains a JUMP_TYPE parameter with value of RETURN, the operation requests a forward to /wdk/return.jsp, returns to the parent form, and removes the nested form history
TimeoutOperation()	The operation requests a forward to /wdk/timeout.jsp

## Form classes

Although the framework supports custom forms, you should customize components rather than forms. The component model allows you to easily configure and customize components and to reuse them elsewhere in your application.

The `com.documentum.web.form` package includes several classes that are used to create a form.

**Form Class** — The Form class extends the Control class. The Form class shares the user's session state between JSP pages. For example, three JSP pages that reference the same form class will share state.

**Note:** Form class objects defined in a JSP page cannot use member variables to hold state across requests.

**FormTag** — The FormTag class generates the HTML for a form. The FormTag class generates HTML that performs the following functions:

- Generates a JavaScript function that calls `setServerForm()` in `events.js`
- Generates hidden input fields such as request ID, scrolling coordinates, action, event handler and arguments, and control.
- Generates JavaScript that sets the scrolling position
- Generates calls to `beginModal()` and `endModal()` that set modality on a window
- Reloads the URL upon user navigation with the **Back** or **Forward** button if the `keepfresh` attribute is set to true

**WebformTag** — The WebformTag class constructs and calls a form processor instance every time a form JSP page is processed by the JSP engine. The webform tag renders a list of script and style sheet inclusions whose content is driven from the resource bundles `com.documentum.web.form.WebformScripts` and `com.documentum.web.form.WebformIncludes`.

## WebformIncludes class

The WebformIncludes class generates included JavaScript files. The WebformIncludes methods are called by the WebformTag class.

**FormIncludeTag** — Forms can be included in other forms if the included form does not contain HTML head and body tags. Use the `forminclude` tag to include a form. For example:

```
<dmf:forminclude name="sub" src="fireEventFormlet.jsp"
 formclass="com.documentum.web.samples.FireEventFormlet">
```

```
nlsclass="com.documentum.web.samples.DemoNls"/>
```

If you set the visible attribute of the `forminclude` tag to `false`, the included form is not rendered.

**IParams** — The `IParams` interface defines a set of constants that are used in HTTP requests to JSP pages or components. The constants are declared as public static final data members. You can use a parameter in a tag class to reference a variable by its `IParams` key.

#### Example 13-29. Using IParams Constants

Variables declared in `IParams` can be used in your class to output HTML. In the following example from the Web Publisher class `ICEEditComponent`, `IParams` constants are appended to the URL:

```
StringBuffer bufUrl = new StringBuffer(lookupString("pages.parsedPage"));

// Append the request Id
bufUrl.append("?");
bufUrl.append(IParams.REQUEST_ID);
bufUrl.append("=");
bufUrl.append(strRequestId);

//Append class name
bufUrl.append("&");
bufUrl.append(IParams.FORM_CLASS);
bufUrl.append("=");
bufUrl.append(this.getClass().getName());
...
bufUrl.append("=");
return makeUrl(getPageContext().getRequest(), bufUrl.toString());
```

The following table describes the constants that are defined in `IParams`.

**Table 13-5. URL parameters**

Parameter	Description
ACTION	Identifies the event raised by a control. Resolves the URL parameter <code>__dmfAction</code> .
CONTROL	Identifies the control that raised an event. Resolves the URL parameter <code>__dmfControl</code> .
DO_VALIDATION	Validation flag for the webform tag. Resolves the URL parameter <code>__dmfDoValidation</code>

Parameter	Description
FORM	Stores the current form instance in the page context (for example, for use in in-line JSP code). Resolves the URL parameter <code>__dmfForm</code> .
FORM_CLASS	Stores the current form class in the page context. Resolves the URL parameter <code>__dmfFormClass</code> .
FORM_HISTORY	Stores the form history setting in the page context. Resolves the URL parameter <code>__dmfFormHistory</code> .
FORM_REQUEST	Stores the form request in the page context. Resolves the URL parameter <code>__dmfFormRequest</code>
FORM_RESPONSE	Stores the form response in the page context. Resolves the URL parameter <code>__dmfFormResponse</code> .
HANDLER	Resolves the URL parameter <code>__dmfHandler</code>
HANDLER_ARGS	Resolves the URL parameter <code>__dmfHandlerArgs</code>
HISTORY_RELEASED	Stores the form history setting in the page context. Resolves the URL parameter <code>__dmfHistoryReleased</code> .
HOST	Stores the external (to the firewall or Web farm) host name used to access the form. Resolves the URL parameter <code>__dmfHost</code> .
HOST_PAGE_URL	Modifies the behavior for a portlet form. Resolves the URL parameter <code>__dmfHostPageUrl</code> .
JUMP_TYPE	Identifies whether to jump, nest or return. Resolves the URL parameter <code>__dmfJumpType</code> . Values: JUMP (exit previous form), NESTED (nest to a form), NOT_SET (jump type not specified), PAGE (sets a page), or RETURN (returns to the parent form).

<b>Parameter</b>	<b>Description</b>
NLS_CLASS	Stores the current NLS class or resource bundle in the page context. Resolves the URL parameter <code>__dmfNlsClass</code> .
REQUEST_ID	ID of the request, used to track form history. Resolves the URL parameter <code>__dmfRequestId</code> .
RES_FOLDER_SESSION_VAR	Identifies the user's selected resource directory for branding. Resolves the URL parameter <code>__dmfResourceFolder</code> .
SERVLET_URL	Identifies the URL of the current servlet. Resolves the URL parameter <code>__dmfServletUrl</code> .
TIMED_OUT	Flag that indicates a session timed out in the request scope. Resolves the URL parameter <code>__dmfTimedOut</code> .
URL	Stores the current page URL in the page context. Resolves the URL parameter <code>__dmfUrl</code> .

## Using the Configuration Service

The configuration service reads configuration settings from XML files at application startup. The contents of configuration files are cached in memory as a DOM for efficient lookup. The application definition DOM is updated if all configuration files contain valid XML, all NLS IDs are located, and there are no duplicate definitions. Runtime errors due to incorrect XML values in component and action definitions report the invalid tag and the name of its configuration file.

The lookup mechanism resolves action, component, and application definitions as well as user-defined definitions.



**Caution:** Changes to XML files are not recognized by J2EE servers. You must restart the server for your changes to take effect. You can refresh component definitions by navigating to the utility page `refresh.jsp`, which is located in the `/wdk` directory.

The following topics describe the configuration service:

- [Configuration service classes, page 479](#)
- [Scope and qualifiers, page 484](#)
- [Context, page 487](#)
- [Configuration service process, page 488](#)
- [Lookup algorithm, page 489](#)

## Configuration service classes

The configuration service is supported by the following classes:

- `ConfigService` ([ConfigService, page 480](#))  
Provides access to config file settings
- `IConfigLookup` ([Configuration lookup, page 481](#))  
Retrieves configuration settings based on context
- `IConfigElement`

Gets and sets the parent element, the element value, the element attribute values, and gets and adds child elements to an element within an individual configuration file. Inheritance is not implemented for this interface.

- `IConfigContext` ([IConfigContext](#), page 480)

Instance of the configuration service that can be used to determine whether a specified context name matches a defined qualifier name

- `IConfigReader` ([Configuration reader](#), page 483)

Loads and reads configuration files

- `IConfigLookupHook` [Configuration lookup hooks](#), page 482

Allows configuration lookup calls to be interrupted and overridden

- `IQualifier` ([Scope and qualifiers](#), page 484)

Maps the component context for the user to scope that is defined within the qualifier class. Every scope that is used in an XML configuration file must have a corresponding qualifier class that implements `IQualifier`.

## ConfigService

At application startup, the configuration service creates a scope rule dictionary. Each entry corresponds to a primary element in the set of configuration files for the application. At runtime, the qualifier class converts component dispatch context and user runtime information into scope values and maps to entries in the scope rule dictionary. The dictionary is continually updated when lookups are performed. For example, when a user requests attributes for an object, the `DocbaseTypeQualifier` class takes the object ID and converts it to type "my\_sop." The configuration services finds the properties component definition containing scope type="my\_sop".

## IConfigContext

The `IConfigContext` interface provides an instance of the configuration service for lookup of qualifiers. The `getConfigContext()` method of `ConfigService` returns the `IConfigContext` interface. You can then query whether a context name matches a qualifier by calling `isContextName()`.

### Example 14-1. Matching arguments to qualifiers

In the following example from `JobExecutionService`, the component execution arguments are passed to the method `getUpdatedContext()` to determine whether they match a defined qualifier:



```

private Context getUpdatedContext(Context context, ArgumentList args)
{
 // add any args to the context that match a config qualifier context name
 IConfigContext configContext = ConfigService.getConfigContext();

 if(context == null)
 {
 context = new Context();
 }

 Iterator iterNames = args.nameIterator();
 while (iterNames.hasNext() == true)
 {
 String strArgName = (String)iterNames.next();
 if (configContext.isContextName(strArgName))
 {
 // add/update to context
 String strArgValue = args.get(strArgName);
 context.set(strArgName, strArgValue);
 }
 }
 return context;
}

```

## Configuration lookup

The `IConfigLookup` interface retrieves configuration elements and values based on context. This lookup implements inherited configuration. This interface provides four methods that look up various types of element values and return the appropriate type or the element itself. You do not need to explicitly import `IConfigLookup` into a component class, because `Component` implements wraps these same methods.

Each method of `IConfigLookup` takes two parameters:

- **String element path:** A period-delimited (.) list of patterns, where each pattern is an element name with an optional attribute **name** and **value** pair appended within square brackets. Three examples:

```

component[id=checkin]
component[id=checkin].pages.start
component[id=properties].contains.component

```

- **(Optional) Context:** A `Context` object can be passed in.

Many components look up values from their XML configuration files using `IConfigLookup` methods. Use these methods to look up `String`, `Boolean`, and `Integer` values from configuration files. The `getContext()` method returns the component's current context.

### Example 14-2. Looking up a configuration string value

The `Doclist` component definition specifies the default object for display as the value of the `<objecttype>` element:

```
<columns>
<!-- default displayed object type(e.g. dm_sysobject) -->
<objecttype>dm_document</objecttype></columns>
```

This value is obtained by the component class in the following way:

```
String strObjectType = lookupString("columns.objecttype");
```

#### **Example 14-3. Looking up a configuration boolean value**

The MyObjects class reads its config setting to find out whether or not to display folders:

```
Boolean bShowFolders = lookupBoolean("showfolders");
if (bShowFolders != null)
{
 m_fIncludedFolders = bShowFolders.booleanValue();
}
```

The configuration service resolves a lookup to the most appropriate component definition using qualifiers. For example, `<component id='checkin'>` resolves to the checkin component. If there are two definitions for checkin qualified by type, for example, `<scope type='dm_sysobject'>` and `<scope type='mytype'>`, the service looks up the definition for the selected object's type. The service then looks for the sub-element `<class>`. If it is not defined in the component definition, the service looks for the sub-element in the definition that was extended, until the value is found. Null is returned if the value is not found or if multiple values exist in the same definition.

## Configuration lookup hooks

You can register lookup hooks for your custom application. A lookup hook interrupts lookup calls. Your lookup hook can override lookup calls or perform some other pre- or post-lookup processing. Each hook must implement `IConfigLookupHook`.

Register the hook class and path in the `com.documentum.web.formext.Environment` properties file. The properties file has the following settings related to configuration lookup classes:

**LookupHookClass#** — Fully qualified class name that implements `IConfigLookupHook`. Replace the `"#"` with an integer starting from 1. The `LookupHookClass`, `Path`, and `Argument` integer should match for each lookup hook class.

**LookupHookPath#** — Scope attribute value. The wild card symbol `"*"` specifies all scopes. In the example below, the preferences hook is used whenever a configuration element path starts with `component.preferences`. If you omit the wild card, the configuration element path must match exactly the specified scope.

**LookupHookArgument** — Specifies whether a path that is passed to the hook methods is absolute or relative to the specified hook path.

**Example 14-4. Registering a lookup hook**

The following example registers a lookup hook for preferences:

```
LookupHookClass.1=com.acme.config.PreferenceLookupHook
LookupHookPath.1=component.preferences.*
LookupHookArgument.1=relative
```

An element path of "component[id=properties].preferences.showAll" is treated as showAll, because the path is relative. The parameters within square brackets are ignored.

**Multiple hooks** — Multiple hooks are supported. If more than one hook matches an element path that is passed in, the configuration service calls each hook until a configuration value is found. The order of the hook entries in the properties file determines the calling order.

The IConfigLookupHook interface specifies three methods, each of which take a configuration element path String parameter and a Context parameter:

**onLookupString(String, Context)** — This method is called when IConfigLookup.lookupString() is called. The method should return a string value, or return null to continue processing.

**onLookupBoolean(String, Context)** — This method is called when IConfigLookup.lookupBoolean() is called. The method should return a boolean value, or return null to continue processing.

**onLookupInteger(String, Context)** — This method is called when IConfigLookup.lookupInteger() is called. The method should return an integer value, or return null to continue processing.

## Configuration reader

The IConfigReader interfaces provides methods to load and read configuration files. The default implementation of this interface is HttpConfigReader, which loads and reads configuration settings from the application root directory on the J2EE server file system.

You can substitute an alternate configuration reader. Specify the reader class in Environment.properties, which is located in /WEB-INF/classes/com/documentum/web/formext. For the value of ConfigReaderClass, provide the fully qualified class name, for example:

```
ConfigReaderClass=com.acme.CustomConfigReader
```

The IConfigReader class specifies the following methods:

**getAppName()** — Returns the name of the application's primary directory as specified in the J2EE deployment descriptor file (web.xml). Specifically, the method returns the value of the element AppFolderName.

**getRootFolderPath()** — Returns the full file system path to the web application root directory.

**loadAppConfigFile(String)** — When passed an application name as a parameter, this method returns a ConfigFile instance. The XML file is loaded from the following path, where strAppName is the name of the application:

```
getRootFolderPath() / strAppName / app.xml
```

**loadConfigFiles(String)** — When passed an application name as a parameter, this method returns an Iterator object of ConfigFile instances within the specified application. The XML files are located from the following path:

```
getRootFolderPath() / strAppName / config
```

## Scope and qualifiers

The configuration service uses the user's context to resolve the appropriate scoped definition and deliver the application, action, or component that is defined for the scope. Scope is defined by a qualifier class that implements the IQualifier interface.

You can use the Context object (refer to [Context](#), page 487) to hold component context data and pass it in to the configuration service. The configuration service will then match the context with a defined qualifier value to find the appropriate action or component definition.

WDK includes the following qualifiers in /wdk/app.xml: repository (docbase), Documentum type, privilege, role, clientenv, application, and version. Webtop adds qualifiers for repository name and user's location in the browser tree. Scope is resolved in the order that qualifiers are specified in app.xml.

- DocbaseNameQualifier
  - Matches the context value "docbase" to the current repository. You can use this qualifier to define a different component UI for each repository. Components that apply to more than one repository, such as Webtop's browsertree or menubar components, cannot be scoped by repository.
- DocbaseTypeQualifier
  - Matches the context values "id" or "type" to the scope "type". For example, the definition, UI, and behavior for attributes defined by type='dm\_sysobject' can be different from the definition, UI, and behavior defined by type='dm\_user'. This

qualifier class also evaluates the object ID to find out whether the object is a reference object and, if so, returns the scope type="foreign".

- **PrivilegeQualifier**

Matches the user's privilege (user\_privilege attribute for dm\_user) to the scope "privilege". For example, users who can create a new group have the creategroup privilege. The newgroup action is scoped to privilege='creategroup'. The list of actions available to all users who do not have the creategroup privilege does not include the newgroup action

- **RoleQualifier**

Matches the context value "role" to the scope "role". Roles must be defined in the repository. For example, the import UI presented to the consumer, defined by role='consumer', can be different from the import UI presented to the administrator, defined by role='administrator'.

- **ClientEnvQualifier**

Matches the context value "webbrowser", "portal", "appintg", or "not appintg" to the scope "clientenv". For example, the login component definition uses the value of appintg and not appintg to present different login pages depending on whether the user is in an Application Connectors environment. The default value is specified as "webbrowser" in app.xml as the value of <application>.<environment>. <clientenv>. For more information on the clientenv qualifier and its use as a filter, refer to [Client environment qualifier, page 53](#).

- **AppQualifier**

Gives all configuration settings an implicit scope of the application in which they reside. The application name is automatically applied to all configuration elements within the application directory. Do not remove this qualifier from your custom list of qualifiers.

- **VersionQualifier**

Specifies a component or action version. Only one version value in a scope is allowed, and the version scope cannot start with "not". Values of the version qualifier are defined in /wdk/app.xml as the value of <supported\_versions>.<version>. The latest version (no version scope value) is implicitly included as a supported version.

A component can extend a non-current version component of the same name, and it will inherit the non-current version's definition. If the custom component or container has a different name from the non-current component that it extends, it must satisfy one of the following requirements:

- Set the scope version explicitly to the same version as the component it extends, for example, <scope version="5.2.5">.
- Set the container <bindingcomponentversion> to the same version as the scope on the component. For example, <bindingcomponentversion>5.2.5</

bindingcomponentversion> on your custom export container means that the component will contain an export component whose scope is version="5.2.5".

- EntitlementQualifier

Evaluates whether Collaborative Edition is enabled for the current repository. If so, the CollaborationService class provides the value "collaboration" for the entitlement scope. Requires a global registry. Refer to *Content Server Installation Guide* for information on global registries.

Qualifier classes define scope values. The map of qualifier values is read into memory when the application first start up.

When a component definition is required during run time, the configuration service converts the user's dispatch context and other runtime information to a scope value. This value is matched to the closest scope that is defined within a configuration file. For example, the user selects of object of a custom type in a list component and selects an action to be performed on the object. The DocbaseTypeQualifier class matches the context value of "objectId" or "type" to scope elements that have a "type" attribute. The action definition for the custom type, if there is one, is dispatched.

Qualifiers have the following features:

- An action or component definition can inherit or override scope

The definition for type=dm\_document inherits the defined parameters and configurable elements for dm\_sysobject unless they are specifically overridden. To override an element, you must provide different content for the element. For example, to require a different set of parameters for the child type, define the <params> element with different <param> elements. If your extended definition has no <params> element, all parameters are inherited.

- An action or component definition can have more than one qualifier value

The list of valid values is comma-separated. For example, <scope type='dm\_document, dm\_folder'> applies to both types of objects and types descended from these types.

- An action or component definition can have more than one qualifier

For example, the newgroup action is scoped to type='dm\_group' and privilege='creategroup'.

- An action or component definition can exclude qualifier values by using the NOT operator

For example, if the definition were scoped to type='dm\_group' privilege='not createtype', any user could create a new group unless they had the privilege createtype.

An application can add a new qualifier element in the application layer app.xml file. In this case, you must specify all WDK qualifiers as well as those of the WDK client

application such as Webtop or Web Publisher, because the qualifiers listed in the top-level application layer override the sets of qualifiers defined in other application layers.

For information on how to configure scope in WDK and client applications, refer to [Scope, page 52](#).

## Context

The Context object can hold data that is specific to the current application context, and that context can be compared with qualifier values or can be used for some other kind of component processing. Some of the uses of the Context object are to hold type, object ID, role, user ID, or configuration settings.

Context information can be passed in a Context object or serialized so that it can be passed as a URL argument or string. To serialize the context, use `Context.serialize()`.

### Example 14-5. Serializing context to a URL

The following example serializes the Context object and encodes it for passing in a URL:

```
Component comp = (Component) getForm().getTopForm();
Context jumpContext = new Context(comp.getContext());
String strContext = context.serialize(jumpContext);
strURLContext = URLEncoder.encode(StringUtil.unicodeEscape(strContext));
```

To decode and deserialize a context that is passed in a URL, use the following syntax:

```
strContext = StringUtil.unicodeUnescape(strURLContext);
Context newContext = Context.deserialize(strContext);
```

### Example 14-6. Saving an object ID in context

The DrillDown component class saves the object ID and object type in a Context object::

```
context.set("objectId", strFolderId);
context.set("type", strType);
```

When the user selects a document in the drill-down view for viewing, the `onClickDocument()` method of the DrillDown class calls:

```
ActionService.execute("view", args, getContext(), this);
```

The view action is scoped by type. For example, see the scoped actions for the type `dm_router` task in `dm_router_task_actions.xml` in `/webcomponent/config/actions`.

The object ID and type are passed in, and the action service verifies that the action is permitted for the given type.

### Example 14-7. Setting a context value

You can set or change the context of a component or control. To change the context within a Java class, use `Context.set()`. In the following example from `DocList`, the method `updateContextFromPath` updates the Context object based on the user's navigation:

```
Context context = getContext();
```

```
...
if (strFolderId == null || strFolderId.length() == 0)
{
 // viewing a docbase
 context.set("objectId", DfId.DF_NULLID_STR);
 context.set("type", "dm_docbase");
}
else
{
 // viewing probably a folder
 IDfSysObject sysobj = (
 IDfSysObject)dfSession.getObject(new DfId(strFolderId));
 String strType = sysobj.getType().getName();
 context.set("objectId", strFolderId);
 context.set("type", strType);
}
```

To set the context within a JSP page, you must override the context that is set when a control is initialized. The following example sets the context for a datafield to `r_object_id`. This would display the appropriate action button for checkin based on document type, assuming you have scoped the checkin component for more than one document type:

```
<dmfx:actionbutton name="Checkin" action="Checkin">
 <dmf:argument name="objectId" datafield="r_object_id"/>
</dmfx:actionbutton>
```

## Configuration service process

The configuration service follows these steps upon initialization:

1. Gets the application name from the deployment descriptor file (web.xml).
2. Loads the application definition from app.xml and any inherited application definitions that are extended in app.xml.
3. Creates qualifiers by enumerating application.qualifiers child elements within the application definition. The qualifier resolves context to a scope element.
4. Loads into document object models (DOMs) the configuration files located within the applications /config directories, implicitly adding the application name to each scope element.
5. Builds a lookup dictionary, which is used to look up primary elements based on dynamic context (refer to [Lookup algorithm, page 489](#))



# Lookup algorithm

The configuration service performs lookups in two phases:

1. Locates the most appropriate primary element
2. Traverses down from the resolved primary element to the requested setting. If the setting is not found, extended primary elements are traversed.

**Resolving the primary element** — When the configuration files are loaded, the configuration service creates an index that contains all primary elements keyed by descriptor and scope. For example, the descriptor could be:

```
component[id=checkin]
```

and the scope could be:

```
type=dm_user,role=*,application=wdk
```

Each scope is listed in order of precedence. The "\*" symbol indicates that the scope value was not defined in the configuration file. If a match is not found, the configuration service key generalizes the key by changing the scope to the parent scope value. The configuration service walks up the parent scope hierarchy until a match is found, or the generic key is used. The generic key matches all scopes defined for the primary element. For example:

```
component[id=checkin]:type=*,role=*,application=*
```

Future lookups are optimized by an update to the index when a match is found.

**Retrieving a configuration setting** — The configuration service starts with the element path that is passed to the lookup method and the primary element that was resolved in the first phase of lookup. The context is not used.

The configuration services walks down the child elements of the primary element to locate the requested value. Filters that surround an element are evaluated to determine whether the element is visible. If the value is not located, the configuration service looks in the parent primary element, specified by the value of the primary element "extends" attribute.



## Customizing actions

The action service implements dynamic filtering of actions based on context.

The action service is described in the following topics:

- [Preconditions, page 491](#)
- [Execution, page 493](#)
- [LaunchComponent execution classes, page 496](#)
- [Providing action NLS strings, page 497](#)
- [Dynamic component launching, page 498](#)
- [Action listeners, page 500](#)
- [Nesting actions, page 503](#)

Refer to [Chapter 4, Configuring Actions](#) for information on configuring actions. For information on the parameters and configurable elements for individual actions, refer to *Web Development Kit Reference Guide*.

## Preconditions

Action precondition classes are called to determine whether an action can be performed. The precondition class determines whether to render an action control as enabled or disabled.

Preconditions are optional in the action definition. If no preconditions are defined, the action will always execute. The precondition class must implement the `IActionPrecondition` interface and the precondition logic for your action. For example, the `CancelCheckoutAction` class, which implements `IActionPrecondition`, gets the object ID and lock owner from the argument list and gets the user name from the session. The precondition class compares the two, and if they are the same, returns true. If the user is not the lock owner, the `queryExecute()` method returns false and the user cannot cancel the checkout.



**Caution:** Make sure that no state (instance variables) is associated with your precondition class. The instance is used as a singleton.

To disable an action, specify `ActionDisablerPrecondition` as the precondition class for the action. Alternatively, you can set the `notdefined` attribute on the action to true. For example, for `dm_folder` scope the versions action is disabled:

```
<action id="versions" notdefined="true"></action>
```

Some precondition classes are used by multiple actions in WDK components:

- Role

The `RolePrecondition` class enables the action for the roles that are specified in a `<precondition>.<role>` element in the action definition.

- Required argument

The existence of a required argument can be assured by the `ArgumentExistsPrecondition` class. Check that the argument has a value using the `ArgumentNotEmptyPrecondition` class.

- Environment

Launch the action in the supported environment using the `EnvironmentPrecondition` class. This precondition is used by WDK for Portlets. For example, the Web workflow manager will not be launched in a portal environment.

Preconditions can affect application performance. For example, preconditions are called for each item in a list component. If there are 10 items and 20 applicable actions, 200 preconditions will be executed before the list is rendered.

**Note:** Do not put error-handling code in preconditions, because preconditions are called to update the UI. Use the precondition to disable the behavior in the UI, or put error handling code in the action execution class.

The precondition class must implement `queryExecute()` to determine whether the precondition has been met. The `queryExecute` method passes in the precondition values in the action definition as `IConfigElement` parameters. The signature for `queryExecute` is:

```
queryExecute(String strAction, IConfigElement config,
 ArgumentList arg, Context context, Component component)
```

where:

- String: Action name
- IConfigElement: Represents the associated `<precondition>` element and subelements. Use this if configuration information is passed to the precondition.
- ArgumentList: List of arguments passed to the action
- Context: The user's current context as defined by the component
- Component: (optional) Caller component. Allows action precondition methods to access the container component instance if necessary.

**Example 15-1. Implementing the action queryExecute() method**

In the following example from `SuspendLifecycleAction`, a class that implements both `IActionPrecondition` and `IActionExecution`, the `queryExecute()` method tests whether the action can be performed:

```
public boolean queryExecute(
 String strAction, IConfigElement config, ArgumentList arg,
 Context context, Component component)
{
 // determine whether the object is locked
 boolean bExecute = false;
 try
 {
 // get lock owner
 String strLockOwner = arg.get("lockOwner");

 // compare
 if (strLockOwner == null || strLockOwner.length() == 0)
 {
 bExecute = true;
 }
 }
 ...
 return bExecute;
}
```

**Note:** Avoid calling `IDfSession.getObject()` or perform queries inside `queryExecute()`. These calls can seriously degrade performance. Most attribute arguments can be retrieved, as they are cached by the initial query on the page rather than from a `getObject()` call. For example, if the page has a databound control to `r_lock_owner`, that attribute value is cached. Your component can check for the existence of the argument value and query only if the argument was not passed. You can also write trace statements that signal when precondition arguments are not passed. This will allow you to determine which custom components are not passing the appropriate arguments.

The precondition class must also implement `getRequiredParams()`, which is called by the action service when the action definition is first loaded. This interface returns an array of parameters that are required by the precondition. For example, the `Checkout` action class `CancelCheckoutAction` returns `objectId` as the required parameter. The same parameters must be defined as required in the XML action definition.

## Execution

The execution class is called by the action service to execute the action when the action preconditions have been met. An execution class must implement `IActionExecution` or extend a base execution class such as `LaunchComponent` (refer to [LaunchComponent execution classes](#), page 496). Action classes in WDK often implement both the precondition and execution interfaces in the same class.



**Caution:** Make sure that no state (instance variables) is associated with your action execution class. The instance is used as a singleton.

The `IActionExecution` interface has two methods:

**execute()** — Returns true if the action has successfully executed. This method is called by the action service after preconditions have been met. For example, the checkout action class returns true for a successful checkout. The parameters are the same as for `IActionPrecondition.queryExecute()` with the addition of a completion arguments Map parameter. The config parameter represents the <execution> elements and its contents.

#### Example 15-2. Implementing action execute() method

In the following example from `SuspendLifecycleAction`, a class that implements both `IActionPrecondition` and `IActionExecution`, the `execute()` method performs the action:

```
public boolean execute(
 String strAction, IConfigElement config, ArgumentList args,
 Context context, Component component, Map completionArgs)
{
 boolean bExecutionSucceeded = false;

 // get the nls bundle for this action
 String nlsprop = config.getChildValue("nlsbundle");

 NlsResourceBundle nlsResBndl = new NlsResourceBundle(nlsprop);
 MessageService msgService = new MessageService();

 try
 {
 String strId = args.get("objectId");
 LifecycleService lifecycleSrvc = LifecycleService.getInstance();

 if (lifecycleSrvc.canSuspend(strId) == true)
 {
 lifecycleSrvc.suspend(strId, null, false, false);
 String strSuccessMessage = nlsResBndl.getString(
 "MSG_SUSPEND_SUCCESS", LocaleService.getLocale());
 msgService.addMessage(
 nlsResBndl, "MSG_SUSPEND_SUCCESS", component, null);
 bExecutionSucceeded = true;
 }
 else
 //error handling, catch block
 }
 return bExecutionSucceeded;
}
```

**Tip:** Do not throw exceptions from `execute()`. Raise non-fatal errors when `execute()` fails, and the errors will display in the message bar. For an example of raising non-fatal errors in a catch block, refer to `com.documentum.webcomponent.library.action.SuspendLifecycleAction`:

```
catch (Exception e)
```

```

{
 ActionExecutionUtil.setCompletionError(completionArgs,
 nlsResBndl, "MSG_SUSPEND_ERROR", component, null, e);
 WebComponentErrorService.getService().setNonFatalError(
 nlsResBndl, "MSG_SUSPEND_ERROR", component, null, e);
}

```

The action can set completion arguments by setting values in the passed completionArgs map. This map is then passed to an action complete listener that is passed in to the action service's execute(...) method. The completion map holds the complete listener (keyed by ActionService.COMPLETE\_LISTENER), allowing the implementation to have access to it if needed. For information on action listeners, refer to [Action listeners, page 500](#).

**getRequiredParams()** — This method returns the parameters required for execution. The same parameters must be defined as required in the XML action definition. The action configuration file must contain at least one required parameter, or the precondition will be ignored. For example, the Checkout action class CancelCheckoutAction returns objectId as the required parameter.

#### Example 15-3. Implementing getRequiredParams()

Your implementation of getRequiredParams() should declare the parameters that are required for the action. In the definition for the suspendlifecycle action, the parameters are declared as follows:

```

<params>
 <param name="objectId" required="true"></param>
 <param name="lockOwner" required="false"></param>
</params>

```

The required parameter is returned by getRequiredParams() in the action definition class as follows:

```

public String [] getRequiredParams()
{
 return new String[] {"objectId"};
}

```

Multiple required parameters are declared in the action class CommentAction as follows:

```

final private static String m_strRequiredParams[] = new String[] {
 "objectId", "contentType"};
public String [] getRequiredParams()
{
 return m_strRequiredParams;
}

```

#### Example 15-4. Passing an action argument value to a component

Actions can pass argument values to a container. The value can be provided in the argument tag or by a datafield. In the following example from the Web Publisher startwpworkflownotemplatecs action definition, an argument and its value are passed to the startwpwftemplatelocatorcontainerclassic container:

```

<execution class="
 com.documentum...LaunchStartWpWfWithChangeSet">
 <container>startwpwftemplatelocatorcontainerclassic
 </container>
 <arguments>
 <argument name='attachmentMode' value='existingchangeset' />
 </arguments>
</execution>

```

The container class `StartWpWorkflowTemplateLocatorContainer` then gets the argument value in the `onInit()` function:

```
m_sAttachmentMode = args.get("attachmentMode");
```

## LaunchComponent execution classes

The `LaunchComponent` class can be used by an action to launch a component that will perform the action after preconditions are met. The `<component>` element in the action definition specifies the component that will be launched. If there is no `<component>` element, the component is assumed to have the same name as the action ID.

The selected object ID, if any, is always passed by the `LaunchComponent` execution class, so you do not need to explicitly pass this argument.

The `LaunchComponent` class is specified as the execution class for an action definition, as follows:

```

<execution class="com.documentum.web.formext.action.LaunchComponent">
 1<arguments>
 <argument name="arg_name" value="arg_value"></argument>
 </arguments>
 2<component>component_name</component>
 3<container>container_name</container>
 4<navigation>jump</navigation>
</execution>

```

- 1 You can pass arguments other than the object ID from the action to the component or container class using the `<argument>` element in the `<execution>`.`<arguments>`.`<argument>` element. In this example, the view action for `dm_wp_task` objects in Web Publisher passes an argument for the container tab ID. The `<argument>` element can contain an alias attribute. This allows an action argument named with the alias name to be passed on to the component.
- 2 An optional `<component>` element specifies the component that will be launched. If a container but no component is specified, the container launches the first component named in the container definition.



- 3 An optional <container> element specifies the container that will launch the component. If no container or component is specified, LaunchComponent attempts to launch a component with the same name as the action.
- 4 An optional <navigation> element specifies the type of navigation to the component that is launched by the action. Valid values are jump, returnjump, and nested. The default is nested, which means that the component will be nested within the component from which the action was called.

The LaunchComponent execution class has special handling when it is used with the ComboContainer or derived container class. Multiple calls to the LaunchComponent execution class in the same request, such as when an action is performed on a multiple selection, results in the combo container being launched once with all of the component instances associated with each call. The LaunchComponent class prepares and passes the parameters that are required by the ComboContainer or derived container class.

If a LaunchComponent action nests a component, the action complete listener is called when the nested component returns. For information on action listeners, refer to [Action listeners, page 500](#). For all other cases, the action completes just before IActionExecution.execute() returns.

For faster performance, the LaunchComponent class does not check object permissions. If your component requires object permissions, such as a custom checkin or import component, use LaunchComponentWithPermitCheck and specify a minimum object permission level. Valid values, from highest to lowest, are delete\_permit, write\_permit, version\_permit, relate\_permit, read\_permit, browse\_permit, and none.

## Providing action NLS strings

To add NLS strings for your custom action, add an <nlsbundle> element to your action definition that points to your custom \*NLSProp.properties file, similar to the following example from the delete action definition:

```
<execution class="
 com.documentum.webcomponent.library.actions.DeleteRenditionAction">
 <nlsbundle>com.documentum.webcomponent.library.delete.DeleteNlsProp
 </nlsbundle>
 ...
</execution>
```

## Dynamic component launching

You can specify which component is launched at runtime based on a dynamic filter in the action definition. Use the `<execution>.<dynamicfilter>` element in the action definition to specify a class that extends `LaunchComponentFilter` and implements the filter. The filter uses an evaluation class that implements `ILaunchComponentEvaluator`. The evaluator class evaluates which component to launch from among the options listed in the filter definition by matching the current context to criteria values in the configuration. For example, the class `ContentTransferLaunchComponentEvaluator` `evaluate()` method returns the appropriate type of content transfer component based on the application environment: An HTTP component is launched for portal environments, and an applet is launched for standalone Web applications.

The contents of the filter element are similar to the following:

```
1<dynamicfilter class=com.mycompany.MyFilter>
 2<option>
 3<criteria>
 4<criterion name="contenttransfer" value="applet"
 evaluatorclass="com.mycompany.ContentTransferLaunchComponentEvaluator"/>
 </criteria>
 5<selection>
 6<component>import</component>
 7<container>importcontainer</container>
 </selection>
 </option>*
</dynamicfilter>
```

- 1 Specifies the filter class and contains two or more options for launching different components from the same action
- 2 Defines an option that will be launched when criteria are met. You must provide at least two options for the filter. Your evaluator class must return a value to the filter class that matches one of the criterion names in your filter definition.
- 3 Contains zero or more `<criterion>` elements whose values must match conditions as implemented by the filter class in order for the component specified in the `<selection>` element to be launched. If this element is empty, the selection is the default selection.
- 4 Defines a criterion that must be matched. The criterion name and value are evaluated by the `dynamicfilterclass` and the corresponding evaluator class is called. The criterion value is evaluated by the criterion evaluator class, which matches the criterion value against its business logic and determines which component selection should be launched.
- 5 Specifies the component and container that will be launched when the criterion is matched

- 6 Specifies the component that will be launched
- 7 Specifies the container that will be launched

#### Example 15-5. Implementing a dynamic filter

An example of the dynamic component filter is the ReferenceEvaluator filter. This filter class determines whether the object is a reference object, referring to an object in another Docbase. If the filter returns a value of true for the isreference criterion, the informInvalidationForReference is launched, which informs the user that the action is not valid for the reference object. If the object is not a reference object, another component is launched to perform the action.

The action framework calls evaluate() on the filter class, passing in the following arguments:

- String strName  
Name of the criterion
- String strAction  
Action ID
- IConfigElement config
- ArgumentList arg  
Arguments passed by the action
- Context context
- Component component  
Component to launch

In the following example from the ReferenceEvaluator class, the evaluate() method uses the objectId argument to determine whether the object is a reference object and returns either true or false for the isreference criterion (error-handling code removed):

```
public String evaluate(String strName, String strAction, IConfigElement config,
 ArgumentList arg, Context context, Component component)
{
 String strValue = "false";
 if (strName != null && strName.equalsIgnoreCase(CRITERION))
 {
 String strObjectId = arg.get("objectId");
 IDfSession session = component.getIdfSession();
 IDfPersistentObject persObj = (IDfPersistentObject) session.getObject
 (new DfId(strObjectId));
 if (persObj.hasAttr("i_is_reference") && persObj.getBoolean("i_is_reference"))
 {
 strValue = "true";
 }
 }
 return strValue;
}
```

```
public static String CRITERION = "isreference";
```

## Action listeners

Actions can have several kinds of listeners:

- Use an action control

All action controls have an `oncomplete` attribute that will call an event handler when the action has completed. Your component class can implement an event handler for the control `oncomplete` event.. The Component class instantiates `IActionCompleteListener` (refer to below) when an event handler is specified for the `oncomplete` attribute of the action control.
- Use `CallbackDoneListener`

Use this listener if your component class launches the action that you are listening to. `CallbackDoneListener` returns the completion arguments to your component.
- Implement `IActionCompleteListener`

Implement this listener if the action you are listening to does not have an associated action control and it is not launched by your component class.
- Implement `IActionListener` (Pre- and post-action listener)

Your component must implement `IActionListener` and two methods: `onPreAction()` and `onPostAction()`. These event handlers will be called by the action service before and after the action is processed, respectively.

The action listener interface `IActionCompleteListener` is called by the action service when an action is completed. This interface exposes one method, `onComplete(String strAction, boolean bSuccess, Map completionArgs)`. The action parameter is the ID of the completed action, the boolean flag is returned by the `IActionExecution.execute()`, and the `Map` is the set of completion arguments that are passed from `IActionExecution.execute()`. Multiselect actions are supported by this listener.

If a `LaunchComponent` action nests a component, the action completes when the nested component returns. For all other action classes, the action completes before `IActionExecution.execute()` returns.

### Example 15-6. Implementing an `onComplete()` event handler

The Component class implements the action listener interface `IActionCompleteListener`, which is called by the action service when an action is completed. This interface exposes one method, `onComplete(String strAction, boolean bSuccess, Map completionArgs)`. The action parameter is the ID of the completed action, the boolean flag is returned by the `IActionExecution.execute()`, and the `Map` is the set of completion arguments that are passed from `IActionExecution.execute()`. Multiselect actions are supported by this listener.

Specify the oncomplete event handler in your JSP page. For example:

```
<dmfx:actionbutton ... oncomplete='onMyActionComplete' />
```

Your component class must implement the event handler for the oncomplete event. The event handler must have the following signature. The method name must match the value of the action control's oncomplete attribute. For example:

```
public void onMyActionComplete(String strAction, boolean bSuccess,
 Map completionArgs)
{
 //code here will be executed when action is complete
}
```

In your custom component JSP page, pass parameters to your action listener in the form of <dmfx:argument> or <dmfx:argument> tags within the action tag. For example:

```
<dmfx:actionimage ...>
 <dmfx:argument name='objectId' datafield='r_object_id' />
</dmfx:actionimage>
```

You can then retrieve the arguments in your component or action class:

```
public void onMyActionComplete(String strAction, boolean bSuccess,
 Map completionArgs)
{
 String newObjectId = (String)completionArgs.get("objectId");
 //code here will be executed when action is complete
}
```

#### Example 15-7. Using CallbackDoneListener

To register your listener class, you must extend the action execution class and override the execute() and getRequiredParams() methods.

Alternatively, you can use the CallbackDoneListener class when you launch your action. In the following example from Web Publisher WpCopy class method onCopy(), the CallbackDoneListener is registered when the copynonwcm action is called:

```
ArgumentList compArgs = new ArgumentList();
compArgs.add("objectId", token);
compArgs.add("folderId", token);
ActionService.execute(
 "copynonwcm", compArgs, getContext(), this, new CallbackDoneListener(
 this, "onReturnFromNonWcmInfoForCopy"));
```

The WpCopy class implements the method whose name is passed to the CallbackDoneListener to handle the action completion. This custom handler must have the same signature as a listener onComplete() method to handle the completion arguments passed by CallbackDoneListener. This example gets the completion arguments in the Map:

```
public void onReturnFromNonWcmInfoForCopy(
 String strAction, boolean bSuccess, Map map)
{
 // if we have return values
 if (map != null)
 {
```

```

try
{
 ...
 String strNumObjects = (String) map.get(
 WpClipboardContainer.KEY_NUM_OBJECTS);
 if (strNumObjects != null)
 {
 for (int i = 0; i < Integer.parseInt(strNumObjects); i++)
 {
 String fileName = null;
 try
 {
 Map retMap = (Map) map.get(
 WpClipboardContainer.KEY_RETURN_VAL + (i + 1));
 ...
 }
 catch (Exception e)
 {
 ...
 }
 }
 }
}

```

### Example 15-8. Implementing IActionCompleteListener

If you need to trap an action that is not launched by your component class or by an action control, you must implement `IActionCompleteListener`.

The following example adds an action listener. The listener is registered by overriding the action execution class. First, create a custom listener class that implements `IActionCompleteListener` and its required methods:

```

import com.mycompany.MyListener;
...
public void onComplete(java.util.Map map)
{
 ...
 m_settingParam = (String) map.get(MyComponent.PARAM);
 ...
}

```

Next, register your listener by extending the action execution class and overriding the `execute()` method, for example:

```

public boolean execute(
 String strAction, IConfigElement config, ArgumentList args,
 Context context, Component component,
 IActionCompleteListener completeListener)
{
 return (super.execute(strAction, args, context, component,
 new MyListener(completeListener, config, args, context, component)));
}

```

### Example 15-9. Implementing IActionListener

The following example from the Webtop message bar component adds the `MessageBar` component as a listener to the session. The component implements `IActionListener`:

```

public void onInit(ArgumentList args)
{
 super.onInit(args);
 if (SessionState.getAttribute(MESSAGEBAR_ACTION_LISTENER) == null)
 {
 ActionService.addActionListener(this, ActionService.SESSION_SCOPE);
 SessionState.setAttribute(MESSAGEBAR_ACTION_LISTENER, new Boolean(true));
 }
}

```

```
}
}
```

The `MessageBar` class in `Webtop` clears the message bar before a new action completes:

```
public void onPreAction(
 String strActionId, ArgumentList args, Context context, Form form)
{
 MessageService.clear(form);
}
```

## Nesting actions

If you need to nest actions, so that the first action calls another action before the action takes place, you can call the second action from the first action's precondition or execution class. You would do this only if the two actions can also be used separately. Alternatively, you can make the second action a listener for the first action or combine the actions into a single action class.

If you need more information from the called action than a simple Boolean return, register your precondition or execution class as an action listener, so that you can get returned completion arguments (refer to [Action listeners](#), page 500).

### Example 15-10. Nesting actions

In the following example from a `DemoteAction` class, which is called from an action button in a business policy UI, the `queryExecute()` method for the demote action first calls an audit action:

```
public boolean queryExecute(String strAction, IConfigElement config,
 ArgumentList arg, Context context, Component component)
{
 // determine whether the object is locked
 boolean bExecute = false;

 try
 {
 ActionService.execute("audit", args, getContext(), this, null);
 // Do the demote action
 }
 ...
}
```





## Customizing Roles

WDK provides two alternative role model plugins to support user roles:

- Client capability role model
- Repository role plugin

For information on how to use and configure these plugins, refer to [Chapter 8, Configuring Roles and Client Capability](#).

The following topics describe role-based customization in the application:

- [Role service APIs, page 505](#)
- [Custom role plugin, page 506](#)
- [Role-based menus, page 507](#)

## Role service APIs

The RoleService evaluates roles for the action and configuration services:

- The action service queries the role service to determine whether an action can be executed based on roles. The role precondition works together with the object's permissions, so that a user can perform actions on the object without having the required role if the user owns the object or is a repository superuser.
- The configuration service queries the role service via the role qualifier to determine which component definition to present to the current user.

The RoleService class has the following methods:

- `getParentRole()`  
Returns the parent role name for the role that is passed as a parameter. This method is used by the configuration service to evaluate scope based on role inheritance. For roles defined as groups, this call returns the parent group. For client capability roles, this returns the super role.
- `isUserAssignedRole()`

Returns true if the user is assigned the role (named in the role parameter) or a base role for the named role. Parameters: String user name, String role name, ArgumentList (object properties used by the action precondition class), Context

- getUserRole()

Returns the role of a user. This method is called by the configuration service to resolve the scope of the role qualifier.

- refresh()

Clears all role caches

## Custom role plugin

If your application needs a different role framework from the client capability model, the Docbase role model, or from a set of roles that include the client capability roles, you can define roles behavior in a custom role plugin. Your plugin will be used instead of the default Docbase role plugin .

To implement a custom role model , use the role model adaptor. Your role model class must implement IRoleModelAdaptor. The default implementation of this adaptor is com.documentum.web.formext.role.DocbaseRoleModel. Your custom role model should be specified as the value of the <rolemodel>.<class> element in your application layer app.xml. For example:

```
<config>
<scope>
 <application>
 <rolemodel>
 <class>com.acme.role.LabRoleModel</class>
 </rolemodel>
 </application>
</scope>
</config>
```

The IRoleModelAdaptor interface defines the same methods as the RoleService class. Your implementation of the method isUserAssignedRole() must determine which role a user should have based on the action argument or component context.

**Note:** Client applications must query the repository for the user's roles. WDK queries the repository for roles and a list of each connected user's roles every 10 minutes.

## Role-based menus

One common customization is role-based menus in which each role is presented with a different menu. The following customization includes all the possible menus in a single custom JSP page and makes the appropriate menu visible based on the user's role:

1. Create a custom menu component definition that extends the Webtop menubar component, and save your configuration file in /custom config:  

```
<component id="menubar" extends="menubar:/webtop/config/menubar_component.xml">
```
2. Using the Webtop menubar JSP page as a model, create a separate dmf:menu control with menu items for each role, and give each menu control a different name. Do not set the visible attribute to either true or false.
3. Extend the Webtop Menubar class and reference this custom class in your custom menubar component definition.
4. In the class onInit() method, get the user name by calling getDfSession().getLoginUserName().

5. Get the user's role from the method of com.documentum.web.formext.role.RoleService:

```
getUserRole(java.lang.String strUsername)
```

6. For the user's role, get the corresponding menu control (substitute the actual control name as the first argument):

```
(Menu)getControl("ctrl_name", com.documentum.web.form.control.Menu).setVisible(true)
```



## Customizing Content Transfer

WDK supports three modes of content transfer that enable the transfer of content between the client and the repository:

- HTTP transfer
- Unified Client Facilities (UCF), used in standalone Web applications
- Applets, supporting customizations based on WDK 5.2.5

You must configure the mode of content transfer that will be supported by your application. The mode is specified in `/custom/app.xml` as are the other global content transfer configuration settings. For more information, refer to [<contentxfer> elements, page 66](#).

**Note:** All content that is specified as **View in Browser** in the user format preferences UI will be delivered via HTTP transfer, regardless of the application default mode of transfer.

The following topics describe content transfer mechanisms.

- [Content transfer modes compared, page 510](#)
- [Unified client facilities \(UCF\), page 513](#)
- [HTTP content transfer, page 529](#)
- [Content transfer listeners, page 531](#)
- [Content transfer service classes, page 532](#)
- [UCF transfer component customization, page 533](#)
- [Content transfer control initialization, page 534](#)
- [Content transfer debugging, page 535](#)
- [Streaming content to the browser, page 537](#)
- [Content transfer progress, page 537](#)
- [Using Pre-5.3 content transfer components, page 536](#)

For information on the content transfer servlets, refer to [Web deployment descriptor \(web.xml\), page 83](#).

The legacy WDK 5.2.5 content transfer applet controls and components are described in *Web Development Kit Reference Guide*.

## Content transfer modes compared

The following table compares feature support for the three modes of content transfer:

**Table 17-1. Feature support in content transfer modes**

Feature	WDK 5.2.5 applet	UCF	HTTP
Download to client	Large applet	Small applet with client-side UCF deployment	No client-side deployment
Drag and drop	Not supported	Supported on IE browser	Not supported
Viewing or editing application	Controlled by browser	Configurable	Controlled by browser
ACS support	Not supported	Supported	Limited support for export or edit; not supported for view with relative links
Progress display	Supported	Supported	Supported only by certain browsers
Preferences	Not supported	Supported	Not supported
Restart interrupted operation	Not supported	Supported	Not supported
Checkout	Supported	Supported	Limited support. User must save and select checkout location.
Edit	Supported	Supported	Limited support. User must save and select checkout location.
Checkin	Supported	Supported	Limited support. User must navigate to saved document.
View	Supported; always transfers content	Supported; does not transfer content if file is up to date on client	Supported; always transfers content
Export	Supported	Supported	Supported

Feature	WDK 5.2.5 applet	UCF	HTTP
Import	Supported	Supported	Limited support. Single file selection at a time, no folder import.
Client xfer tracing	Supported	Supported	Not supported
Server xfer tracing	Supported	Supported	Supported
File compression	Limited support (on or off)	Supported, with configurable exceptions	Turn on HTTP compression in web.xml
XML application	Supported	Supported	Import single file against Default XML Application
Virtual document	Supported	Supported	Root only

The configuration settings for UCF, HTTP, and WDK 5.2.5 applet modes are mapped in the following tables:

**Table 17-2. How client configuration settings map in content transfer modes**

Setting	Applet (app.xml)	UCF (client.config.xml)	HTTP
Temp working directory for upload/download	contentlocationXXX	temp.working.dir	None
Checked out files	checkoutlocationXXX	checkout.dir	User must save file, then check in from file
Exported files	viewedlocationXXX	export.dir or user choice	Browser-specific UI to select a download location
Viewed files	viewedlocationXXX	viewed.dir	Browser temporary internet files location, for example, %user.home%\Local Settings\Temporary Internet Files
User location (base for other locations)	userlocationXXX	user.dir (defined in config file)	None

Setting	Applet (app.xml)	UCF (client.config.xml)	HTTP
Registry file location	registrylocatio-nunix	registry.file	None
Registry mode	registrylocatio-nunix	registry.mode	None
Log file location	wdk.log	logs.dir	None
Tracing/debug	Limited: debug stops cleanup of temp files on client	tracing.enabled	None
File polling	None	file.poll.interval	None
Buffer and chunk size	bufferize uploadchunksize	None	None
Removal of viewed files	housekeepingXXX	UCF operation reads registry key	None

XXX denotes settings that have more than one value.

**Table 17-3. How server configuration settings map in content transfer modes**

Setting	Applet (app.xml)	UCF (server.config.xml)	HTTP
Temp working directory for upload/download	server. contentlocation*	server. contentlocation*	server. contentlocation*
Tracing/debug	Limited: debug stops cleanup of temp files on server	tracing.enabled	Limited: debug stops cleanup of temp files on server
Polling for config file change	None	file.poll.interval	None
Log file location	wdk.log	DFC logging file in log4j.properties	wdk.log
File compression	inlinecompression*	compression. exclusion.formats	None



## Unified client facilities (UCF)

UCF is a lightweight client-based service that transfers content between the client, application server, and Content Server. UCF performs the following functions:

- Standardizes content handling across infrastructure and applications
- Simplifies XML and compound document processing in a Web environment by decoupling DFC and WDK and by providing an open framework for content analysis
- Improves reliability, maintainability, and performance of WDK content transfer
- Requires no DFC on the client

**Note:** UCF content transfer is not used for import when the user has selected accessibility mode. HTTP content transfer is used in accessibility mode.

Information about UCF in the WDK framework is provided in the following topics:

- [UCF on the client, page 513](#)
- [Configuring the UCF client, page 514](#)
- [Configuring UCF client path substitution, page 517](#)
- [Configuring UCF support for unsigned or non-trusted SSL certificates, page 518](#)
- [UCF on the application server, page 520](#)
- [Configuring the UCF application server, page 521](#)
- [Configuring UCF support for chunked transfer encoding, page 522](#)
- [UCF troubleshooting, page 524](#)
- [Windows client registry in content transfer, page 527](#)
- [UCF process, page 525](#)

## UCF on the client

A lightweight installation applet silently installs UCF on the client. This applet does not download and install on the client, so it does not require special permissions for the user. The applet activates UCF, which downloads and installs UCF components (not applets) on the client. The applet automatically checks with the UCF server to see whether it requires any missing or updated client services.

The UCF client does not require DFC. It does require Java 1.4.2 or higher on the client machine. For Windows clients, if the appropriate version of Java is not found, then the UCF installer silently installs a private copy of the JRE for the use of UCF only. The private JRE is not used by the browser.

UCF sessions are short-lived. At the end of the request, the unused session is disconnected. If the session is not released at the end of the request, it is disconnected upon the next `release()` call.

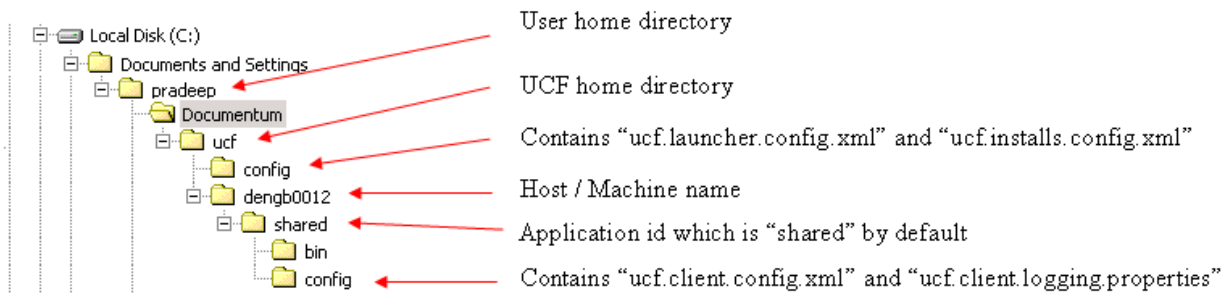
For information on configuration default file locations on the client, refer to [Configuring the UCF client, page 514](#).

## Configuring the UCF client

You can configure default settings for client locations and SSL support in the UCF configuration files. Client location settings are overridden by locations specified in the client registry. Refer to [Windows client registry in content transfer, page 527](#) for details on the registry keys. Both the default settings and the registry settings for locations on the client are overridden by user preference after UCF has been installed on the client.

UCF client default settings are configured in `ucf.installer.config.xml`, which is located in `/wdk/contentXfer`. The contents of this file are used to write a platform-specific config file on the client, `ucf.client.config.xml`. The location of this file on the client is specified in `ucf.installer.config.xml`, for example, the user's OS home directory, for example, `C:\Documents and Settings\pradeep\Documentum\ucf\DENG0012\shared\config`. (This location is configurable.) A typical installation is diagrammed below. (The locations are configurable.)

**Figure 17-1. UCF sample client configuration mapping**



The following table describes the configurable settings in `ucf.installer.config.xml`. (Not all settings are configurable.):

Table 17-4. UCF client configuration settings

Element	Description
<platform>	Specifies platform details. The combination of os and arch values must be supported by the installed WDK release version. Valid values for attribute os: all   windows   mac   solaris   hp-ux   aix   linux Valid values for attribute arch:all   x86   ppc   sparc   pa_risc   power   power_rs   i386. The combination os="all" and arch="all" sets defaults for all platforms. The values within <platform> override these defaults.
<defaults>	Specifies the default values to be used if the installer cannot determine a value. These values are overridden by settings in <defaults> element within a platform-specific element (<platform>).
<ucfHome>	Location for the UCF runtime on the client. Refer to <a href="#">Configuring UCF client path substitution</a> , page 517.
<ucfInstallsHome>	Location for components installed by UCF on the client. Refer to <a href="#">Configuring UCF client path substitution</a> , page 517.
<configuration>	Contains <option> elements that configure default content transfer file locations on the client
<option>.<value>	All path (option name ends in .dir) must be absolute or begin with a path substitution variable (refer to * below). user.dir: Sets base location for export.dir, checkout.dir, viewed.dir, temp.working.dir, and logs.dir export.dir: Default location for exported files (UI allows user to select location during export, saved as preference) checkout.dir: Location for checked out files viewed.dir: Location for viewed files temp.working.dir: Location for temporary

Element	Description
	upload download of files, cleaned up after user specifies the destination logs.dir: Location of client log file, used when client tracing is enabled in tracing.enabled option
Option names	
registry.mode	Type of registry that tracks checked out files. Valid values windows (default, Windows registry)   file (all non-Windows platforms) For file mode, registry file location will be written into client config file. All platforms except Macintosh: user.dir\documentum.in. Macintosh: user.dir\registry.xml
tracing.enabled	Turns on client UCF tracing to a log file ucf.trace*.log where * is a number appended to the log name.
file.poll.interval	Interval in seconds (non-negative) to poll for changes to UCF configuration files
https.host.validation	Set value to false and persistent to false to stop server validation of SSL certificates. (User is still required to accept certificate.) Default value=true. Refer to <a href="#">Configuring UCF support for unsigned or non-trusted SSL certificates</a> , page 518.
https.truststore.file	Specifies the location of the trust store file containing one or more self-signed or untrusted SSL certificates. Path must be valid for all users. Refer to <a href="#">Configuring UCF support for unsigned or non-trusted SSL certificates</a> , page 518

Element	Description
https.truststore.encrypted.password	Specifies the trust store password. Default="changeit". You can use the UCF password encryption utility to provide an alternative password. Set the value attribute to the encrypted password and the persistent attribute to false. Refer to <a href="#">Configuring UCF support for unsigned or non-trusted SSL certificates, page 518</a>
cipher.name	Name of cipher used to encrypt truststore password. Refer to <a href="#">Configuring UCF support for unsigned or non-trusted SSL certificates, page 518</a>
cipher.secret.key	Cipher key bytes encoded in base64. Refer to <a href="#">Configuring UCF support for unsigned or non-trusted SSL certificates, page 518</a>
cipher.secret.key.algorithm	Name of algorithm to use for cipher key. Refer to <a href="#">Configuring UCF support for unsigned or non-trusted SSL certificates, page 518</a>
<platform>.<java>	Sets the Java version and location for the platform
<platform>.<nativelibs>	Sets the version and location of platform-specific native libraries used by UCF

**ucf.installs.config.xml** — The file `ucf.installs.config.xml` describes all of the UCF installations on the machine. This file is located in the UCF installation home, which is typically `user_home/Documentum/ucf/config/`. In the same directory is `ucf.client.logging.properties`, which describes the logging (not tracing) options that control Java logging behavior for the UCF client. For more information on UCF logging, refer to [UCF logging, page 523](#).

## Configuring UCF client path substitution

The UCF client configuration file can be configured to provide locations for client content using path substitution variables. UCF client default settings are configured in `ucf.installer.config.xml`, which is located in `/wdk/contentXfer` in your installed WDK application.

The path to location variables in this configuration file can begin with one of the following substitution variables:

- `$java{...}`: Any Java system property, for example, `$java(user.home)`
- `$env{...}`: Any environment variable, for example, `$env(USERPROFILE)` in Windows or the equivalent on other platforms
- `$ucf{...}`: Any UCF configuration option, for example, `$ucf{user.dir}`

Path substitutions can be mixed, with more than one substitution within a string.



**Caution:** All substitutions must resolve to one path per user. For example, on Windows, `$env{USERPROFILE}/Documentum/ucf` is valid, but `$env{HOMEDRIVE}/Documentum/ucf` is not. This restriction is not enforced by the UCF installer and must be planned for by the application administrator.

Substitution paths must be defined before they are used. For example, if `user.dir` is defined first, it can then be used in the second path as shown below:

```
<option name="user.dir"><value>$java{user.home}/Documentum
 </value></option>
<option name="export.dir"><value>$ucf{user.dir}/Export
 </value></option>
```

The following substitution is invalid because the variable `user.dir` is used before it is defined:

```
<option name="export.dir"><value>$ucf{user.dir}/Export...
<option name="user.dir"><value>$java{user.home}/Documentum...
```

**Tip:** Generally, `<ucfHome>` and `<ucfInstallsHome>` have the same value. However, if the user home directory is on a network share, performance may be improved if UCF binaries are installed in a local directory rather than in the home directory. In this case, change `<ucfHome>` to point to a local directory. This directory must be unique for each OS user.

## Configuring UCF support for unsigned or non-trusted SSL certificates

If your enterprise has configured the application server to use an SSL certificate that is issued by a certifying authority (CA) not trusted by Java or to use a self-signed certificate, you have two options:

- Configure UCF to switch to non-server validation. This means that the certificate will not be validated against the Java trust store. The default is server validation against the Java trust store, so you must explicitly change this setting to support self-signed certificates or certificates from untrusted CAs. The user must accept the certificate before UCF can work over SSL.

- Install a trust store on each client machine containing the self-signed or non-trusted certificate and configure the UCF installer configuration file to locate and access the trust store. Subsequent updates to the VM will not require an update of trust store file.

**Configuring non-server validation mode** — In non-server validation mode, UCF will override the Java default implementation of Trust Manager to allow UCF client to automatically trust certificates used for SSL authentication. Modify `ucf.installer.config.xml`, which is located in `/wdk/contentXfer` in your installed WDK application.. Add the following option within the `<configuration>` element:

```
<option name="https.host.validation" persistent="false"><value>>false
</value></option>
```

**Configuring server validation mode** — Install a truststore in each client machine containing one or more self-signed or untrusted certificate. You can install this to a pre-defined location, such as `$java{user.home}/Documentum/.truststore` where `.truststore` is the name of the file that contains the certificates.

Set the location in `ucf.installer.config.xml` in the Web application by adding the following option element to the `<configuration>` element:

```
<option name="https.truststore.file"><value>path_to_client_truststore
</value></option>
```

The file path to the truststore file must be valid for all clients.

You can also modify the UCF configuration on each client after UCF has been installed on the client. Navigate to the client UCF home location. This can be located by examining the value of `<ucfHome>` in the Web application file `ucf.installer.config.xml`, located in `/wdk/contentXfer`. Add `/machine_name/shared/config` to this path, and you will locate the file `ucf.client.config.xml`. Edit or add the option element `https.truststore.file` as shown above.

**Encrypting a trust store password** — To access the certificates trust store, UCF must have the trust store password. The default used by UCF is "changeit". You can encrypt your own password and store the encrypted password in the UCF installer configuration file.

To encrypt a password, use the default Cipher class in UCF. This utility has the following syntax:

```
java -cp ...com.documentum.ucf.common.util.spi.BaseCipher password
[cipher_name key_algorithm] [password_encoding]
```

Your classpath (`-cp ...`) must contain references to `ucf-client-impl.jar` and `ucf-client-api.jar`. (These APIs are present in the WDK application directory `/wdk/contentXfer`.)

Examples:

```
com.documentum.ucf.common.util.spi.BaseCipher "my password"
```

```
com.documentum.ucf.common.util.spi.BaseCipher "my password" UTF-8
com.documentum.ucf.common.util.spi.BaseCipher "my password"
 DES/ECB/PKCS5Padding DES
com.documentum.ucf.common.util.spi.BaseCipher "my password"
 DES/ECB/PKCS5Padding DES UTF-8
```

The output of the utility will be similar to the following:

```
cipher.name: DES/ECB/PKCS5Padding
cipher.secret.key: 00V8MsKbeto=
cipher.secret.key.algorithm: DES
Encrypted password (e.g. https.truststore.password): VIGQdGy1YAQ=
Password encoding (e.g. https.truststore.password.encoding): UTF-8
```

Copy the encrypted password and encoding to their respective <option> elements in `ucf.installer.config.xml`.

**Note:** Some algorithms in the Java Cryptography Extension Reference Guide (Appendix A) are not supported. The supported algorithm must take the key as a byte array. Stronger algorithms can be used and deployed to the JRE `lib/security` directory.

## UCF on the application server

The UCF server runs in the application server. It is deployed as part of the application, that is, within the WAR file.

WDK UCF components are paired as one container and one component per content transfer operation. The container has an associated service class that orchestrates the DFC/UCF/WDK interaction and progress information. The service class is invoked and executed asynchronously by the container class, using the WDK asynchronous job framework. Progress is reported by a job progress component. Control is returned to the container when the job completes or user interaction is required.

The content transfer component is associated with a service processor that propagates values from the UI to the DFC operation and operation nodes. To support UCF content transport, a content transfer component definition must contain a <ucfrequired/> element. The element turns on UCF for all pages by default. UCF can be turned off for specific events or JSP pages within a component in the following optional elements within a component definition:



<ucfrequired>	Contains zero or more <events> and/or <pages> elements
<events>	Contains one or more <event> elements that correspond to events in the component class, for example, onInit. The component class method must be the value of the <event> name attribute, and the enabled attribute must be set to false to bypass UCF for the event. If the <ucfrequired> element is present and UCF is disabled for events, one or more pages must have UCF enabled.
<pages>	Contains one or more <page> elements that correspond to JSP pages in the component definition, for example, start. The page element (<pages>.<start> in this example) in the component definition must be the value of the <page> name attribute, and the enabled attribute must be set to false to bypass UCF for the page.

The content transfer service, processor, and transport classes handle UCF, HTTP, and applet-based content transfer. Those classes are described in [Content transfer service classes](#), page 532.

For information on configuring UCF server settings, refer to [Configuring the UCF application server](#), page 521.

## Configuring the UCF application server

You can configure default settings for server file locations in the UCF server configuration file. The server configuration file `ucf.server.config.xml` is located in `/wdk/src`. After installation on the application server, this file is located in `/WEB-INF/classes`. The following table describes server configuration options in `ucf.server.config.xml`:

**Table 17-5. UCF application server configuration settings**

Element	Description
temp.working.dir	Location for temporary upload download of reads and writes, cleaned up content transfer has completed. Not used by WDK, which uses server.contentlocation* in app.xml.
tracing.enabled	Turns on server UCF tracing to a DFC log file trace.log in the location specified by log4j.properties
file.poll.interval	Interval in seconds (non-negative) to poll for changes to UCF configuration files
compression.exclusion.formats	Specifies formats to be excluded from compression during file transfer.
http11.chunked.transfer.encoding	Sets chunked content transfer encoding to HTTP 1.1 chunking (default) or UCF alternative chunking (for certain reverse proxy environments). Valid values: enabled (default)   disabled (UCF alternative)   enforced (forces HTTP 1.1. chunking, for debugging and testing only). Refer to <a href="#">Configuring UCF support for chunked transfer encoding, page 522</a> for information.
alternative.chunking.buffer.size	Server informs the client of the buffer size in bytes for UCF alternative chunking. (Does not apply to HTTP 1.1 default chunking.) Default = 2M. Value must be an integer; with units in bytes, kilobytes(K), or megabytes (M), for example, 1M or 512K (no space). Refer to <a href="#">Configuring UCF support for chunked transfer encoding, page 522</a> for information.

## Configuring UCF support for chunked transfer encoding

Some environments with external Web servers that servr as forward or reverse proxy servers do not support native HTTP 1.1 chunking. For these servers, you must configure

ucf.server.config.xml, which is located in /WEB-INF/classes. To disable HTTP 1.1 chunking and use UCF alternative chunking, edit the following options:

```
<option name="http11.chunked.transfer.encoding"><value>disabled
</value></option>
<option name="alternative.chunking.buffer.size"><value>1M
</value></option>
```

For performance tuning, you can change the default buffer size for UCF alternative chunking. Buffer sizes: Default = 2M. Value must be an integer; with units in bytes, kilobytes(K), or megabytes (M), for example, 1M or 512K (no space).

## UCF logging

WDK provides logging at the content transfer component level. For information on this logging, refer to the section in this document entitled "Content transfer debugging."

Server-side UCF uses DFC's infrastructure for logging. The logs go to the location specified in the log4j.properties file. UCF server API tracing is turned on in ucf.server.config.xml:

```
<option name="tracing.enabled">
 <value>true</value>
</option>
```

Client-side UCF uses the Java logging infrastructure similar to that of log4j. The location is configured in ucf.client.logging.properties file, which is typically found in *user\_home/Documentum/ucf/config/*. The default log level is set to WARNING. For a more granular level during troubleshooting, set it to FINE or FINER. Typical log contents are shown below:

```
Nov 3, 2005 9:22:54 AM com.documentum.ucf.client.logging.impl.UCFLogger fatal
SEVERE:
Unrecoverable stream error:
 com.documentum.ucf.common.configuration.ConfigurationException:
 java.io.FileNotFoundException: ucf.server.config.xml
 com.documentum.ucf.common.transport.TransportStreamException:
Unrecoverable stream error:
 com.documentum.ucf.common.configuration.ConfigurationException:
 java.io.FileNotFoundException: ucf.server.config.xml
 at com.documentum.ucf.client.transport.impl.ClientReceiver.getRequests(
 ClientReceiver.java:51)
 at com.documentum.ucf.client.transport.impl.ClientSession.handshake(
 ClientSession.java:414)
 at com.documentum.ucf.client.transport.impl.ClientSession.run(
 ClientSession.java:176)
```

Tracing is enabled in ucf.client.config.xml. Trace files (ucf.trace\*.log) go to the location specified, as in the example shown below:

```
<option name="logs.dir">
 <value>C:\Documentum\logs</value>
```

```
</option>
<option name="tracing.enabled">
 <value>true</value>
</option>
```

Typical trace output is shown below:

```
Mar 25, 2005 11:23:44 AM com.documentum.ucf.client.logging.impl.UCFLogger trace
FINER:
.com.documentum.ucf.common.configuration.IClientConfigurationService.
 getConfiguration
 [started]
Mar 25, 2005 11:23:44 AM com.documentum.ucf.client.logging.impl.UCFLogger trace
FINER:
.getConfiguration [finished]
Mar 25, 2005 11:23:44 AM com.documentum.ucf.client.logging.impl.UCFLogger trace
FINER:
.com.documentum.ucf.common.configuration.IConfiguration.getOptionValue
 [started]
Mar 25, 2005 11:23:44 AM com.documentum.ucf.client.logging.impl.UCFLogger trace
FINER:
.getOptionValue [finished]
Mar 25, 2005 11:23:44 AM com.documentum.ucf.client.logging.impl.UCFLogger trace
FINER:
.com.documentum.ucf.client.transport.impl.ClientSession.findArgIndex
 [S] [started]
Mar 25, 2005 11:23:44 AM com.documentum.ucf.client.logging.impl.UCFLogger trace
FINER: .findArgIndex [finished]
```

## UCF troubleshooting

Refer to [UCF logging, page 523](#) for instructions on turning on and interpreting UCF logging.

**Problem scenario: "Initializing plug-in..." page hangs** — Troubleshooting steps:

1. See if UCF client is installed correctly, especially configuration files. Look at paths in UCF client configuration file (refer to [Configuring the UCF application server, page 521](#)) to make sure user has permissions on the target directories.
2. Open the browser Java console look for "invoked runtime: ... connected, uid: ..". "Invoked runtime" indicates successful launch of the process. A UID indicates successful connection to the UCF server.

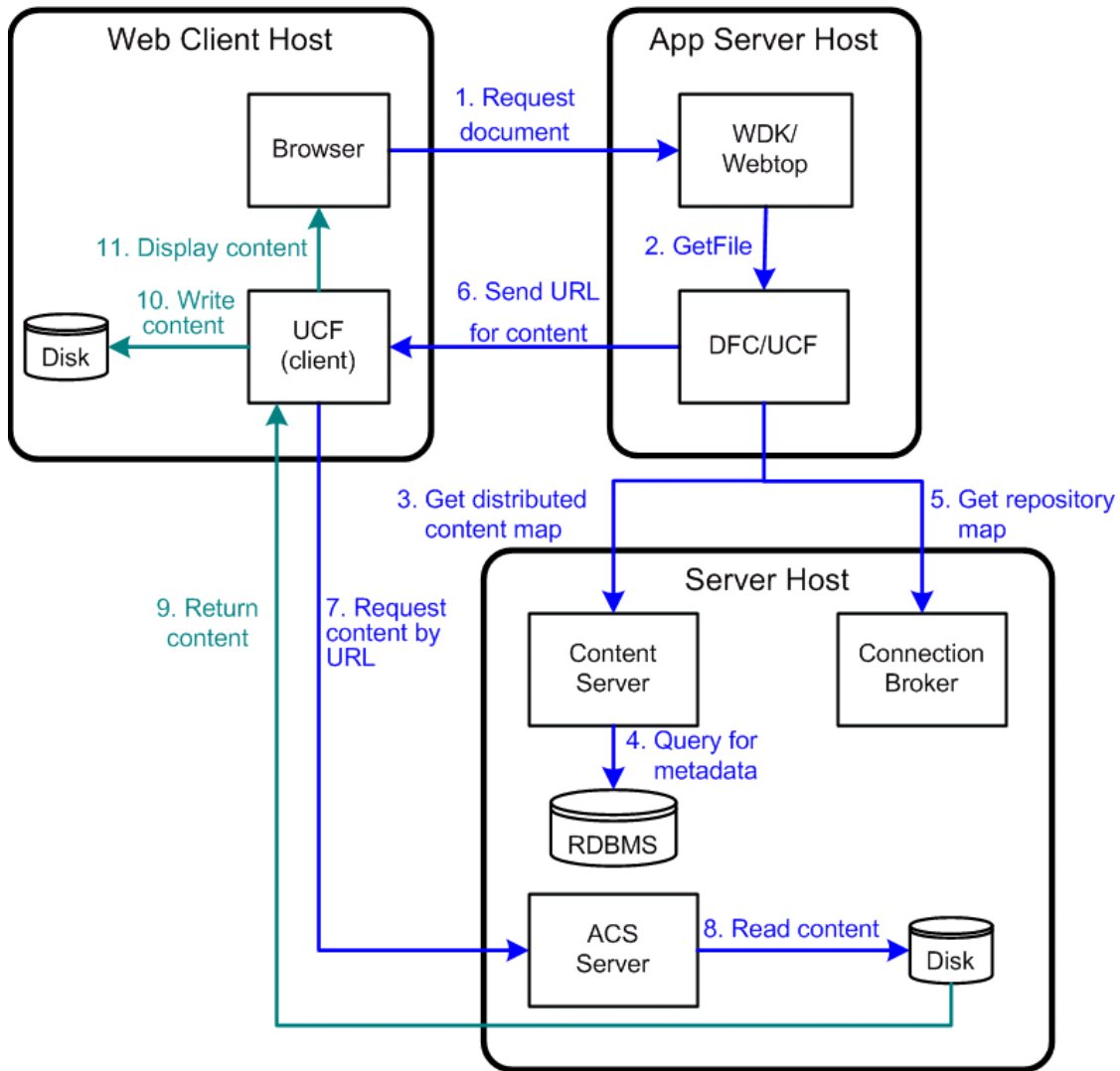
**UCF client runtime is not responding** — Troubleshooting steps:

1. See if the process from the launch command is running: Open the browser Java console look for "invoked runtime: ... connected, uid: ..." A UID indicates successful connection to the UCF server.
2. Are there any errors on the UCF server side? Check the application server console.
3. Restart the browser and retry the Webtop operation.
4. Kill the UCF launch process and retry the Webtop operation.

## UCF process

Requests for content transfer are passed from the client to the WDK application and then to UCF through the HTTP client-server mechanism. The components are diagrammed below:

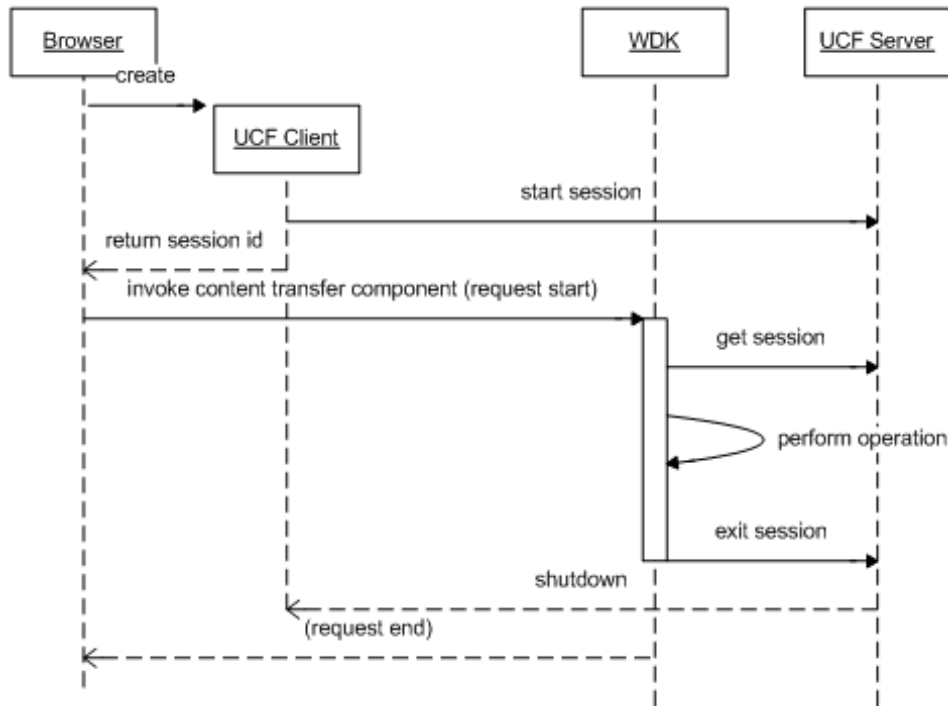
Figure 17-2. UCF client-server process



In UCF mode, all content transfers are handled by the UCF client runtime process, outside of the browser process. The UCF process is instantiated for each browser request which requires UCF for its processing

The sequence of interaction between the browser, server, and UCF runtime is shown in the following diagram:

Figure 17-3. UCF client-server session management



## Windows client registry in content transfer

WDK 5.2.5 applet content transfer components, and some UCF content transfer components, use Windows registry keys to read the locations for content transfer files on the client. The registry is also used to record the name and full path to viewed or checked out files. The settings in these keys override the default settings in the UCF client configuration files: `ucf.installer.config.xml`, which is used by the client installer to create the client configuration file `ucf.client.config.xml`. If the user does not have existing locations specified in the Documentum registry key, the default settings from this configuration file will be used. Users can then set location preferences, which then override both the default configuration settings and the registry settings.

The client registry keys are summarized below. Keys are relative to `HKEY_CURRENT_USER (HKCU)\SOFTWARE\Documentum`. If the registry key does not exist, the location specified in the UCF client configuration file is used. Refer to [Configuring the UCF application server, page 521](#) for details on the client defaults.

**Table 17-6. Content transfer registry keys used by content transfer applets**

Key Name	String	Purpose
\Common	CheckoutDirectory	Path to checkout directory on client. If not found, defaults to path specified in <contentxfer>.<client>.<checkoutlocationwindows> (applets) or value in UCF client config file (UCF).
\Common	ExportDirectory	Path to directory for exported and viewed files on client. If not found, defaults to path specified in <contentxfer>.<client>.<viewedlocationwindows> (applets) or or value in UCF client config file.
\Common\InlineDescendants		(Applets only) Each string specifies the path to a checked out file that is a viewable descendant of the root XML file. If a chunk is read-only, its path is recorded in the ViewFiles key.
\Common\ViewFiles		Specifies the path to a file that is a file downloaded for viewing and other information (applets and UCF)
\Common\WorkingFiles		Specifies the path to a file that is checked out to the checkout directory (applets and UCF)
\HouseKeeping	LastHouseKeeping	Specifies the date of last housekeeping (applets only). DFC modifies this date after cleaning up viewed files.



# HTTP content transfer

All content transfer operations are available via HTTP, with some limitations (refer to below). HTTP transfer mode uses the transfer mechanism that is defined in the following standards:

- HTTP/1.1, RFC 2616 (sections 3.6.1, 14.17)
- Form-based file upload, RFC 1887
- Content-Disposition header, RFC 2183

**Note:** Browser popup blockers interfere with content download because HTTP download opens a new window to display downloaded content. You can try turning off the popup blocker or adding the WDK server to the trusted sites in the browser.

An HTTP file upload submits the file content in multipart/form-data encoded HTML forms. The request is filtered by the RequestAdaptor filter. This filter, specified in web.xml, wraps requests of type `HttpServletRequest` in `MultipartHttpRequestWrapper` in order to facilitate separation of the multipart content from the request parameters.

The filebrowse control in upload mode ensures that the parent form is rendered with multipart encoding set. When the user selects a file, the file as well as its path become part of the control state.

HTTP file download sends the file content inline in HTTP responses. Header information in the response allows the browser to reconstruct the original file name. The mime-type in the response header allows the browser to select the appropriate viewer or editor application for the content. The view component uses the virtual link handler to deliver content, ensuring that relative links in the content are resolved by the browser.

**Limitations** — HTTP content transfer is supported for XML files but only for a single file used with the Default XML Application. For virtual documents, only the root (parent) file is transferred. The browser handles the launch of viewing and editing applications.

The checkout directory is not configurable. To check out a document, users must select Edit, then either save the document or open and save it. On checkin, the user must navigate to the location in which the document was saved.

User preferences are not supported in HTTP mode.

## **Example 17-1. Getting multi-part upload files in the request**

Files that are uploaded via HTTP are available from `HttpTransportManager.getUploadedFiles(paramName)`. The file name is in `ServletRequest.getParameter(paramName)` or `ServletRequest.getParameterValues(paramName)`.

The following example uses the filebrowse control to get the file paths on the client machine and provide the path to the HTTP import component and container.

The JSP page that gets the content contains the filebrowse control as follows. This table row can be repeated to get multiple files in the same upload:

```
<table><tr>
 <td align="left" valign="top" width="100%">
 <dmf:filebrowse name="filebrowse" cssclass="
 defaultFilebrowseTextStyle" size="50" fileupload="true"/>
 </td>
</tr></table>
```

The code that gets the files and their metadata and displays the metadata in a JSP page is shown below:

```
<table border="0" cellpadding="1" cellspacing="0" width='100%'>
<%
 HttpTransportManager manager = HttpTransportManager.getManager();
 for (Enumeration e = manager.getUploadedFileParameterNames(
); e.hasMoreElements();)
 {
 String param = (String) e.nextElement();
 String[] fileNames = request.getParameterValues(param);
 File[] files = manager.getUploadedFiles(param);
 for (int i=0; i<files.length; i++)
 {
%>
 <tr>
 <td style="font-variant: small-caps; font-weight: bold;">
 param name: </td>
 <td style="white-space: nowrap">
 <%=param%></td></tr>

 <tr>
 <td style="font-variant: small-caps; font-weight: bold;">
 client filename: </td>
 <td style="white-space: nowrap">
 <%=fileNames[i]%></td></tr>

 <tr>
 <td style="font-variant: small-caps; font-weight: bold;">
 path on server: </td>
 <td style="white-space: nowrap">
 <%=files[i].getAbsolutePath()%></td></tr>

 <tr>
 <td style="font-variant: small-caps; font-weight: bold;">
 size: </td>
 <td style="white-space: nowrap">
 <%=files[i].length()%></td>
 </tr>
 }
 }
%>
</table>
```

## Content transfer listeners

Most UCF content transfer containers set a return value when the task has completed. The names of those return values are exposed as public constants in each container:

**Table 17-7. UCF content transfer result variables**

Variable	Description
CancelCheckoutContainer. NEW_OBJECT_IDS	Returns a map of object IDs of the objects before checkout
CheckinContainer. NEW_OBJECT_IDS	Returns a map of new object IDs for the objects checked in
CheckoutContainer. NEW_CLIENT_PATHS	Returns a map of object IDs to file paths for the corresponding files on the client
ExportContainer. NEW_CLIENT_PATHS	Returns a map of object IDs to file paths for the corresponding files on the client
ImportContentContainer. NEW_OBJECT_IDS	Returns a list of object IDs for new objects ids for the imported files. Refer to <i>Web Development Kit and Applications Tutorial</i> for an example that gets the new object IDs.

You can retrieve values or perform some other operation after an action has finished by instantiating a `CallbackDoneListener` in your component class. The following example retrieves checkout return values:

```
public void someMethod()
{
 ...
 ActionService.execute("checkout", args, this.
 getContext(), this, new CallbackDoneListener(
 this, "onReturnFromCheckout"));
}

public void onReturnFromCheckout(
 String strAction, boolean bSuccess, Map map) {
 ...
 HashMap hmNewClientPaths = (HashMap) map.get(
 CheckoutContainer.NEW_CLIENT_PATHS);
 if (hmNewClientPaths != null)
 {
 for (Iterator iter = hmNewClientPaths.entrySet(
).iterator(); iter.hasNext();)
 {
 Map.Entry e = (Map.Entry) iter.next();
 String objectId = (String) e.getKey();
 String clientPath = (String) e.getValue();
 ...
 }
 }
}
```

```
 }
 }
}
```

## Content transfer service classes

The content transfer service layer is component of three parts: a content transfer service class, a service processor, and a content transport class. Each content transfer operation involves one service class, one transport class, and multiple processors. These service classes interact with the component and container classes to perform the content transfer task.

The container transfer container definition specifies a service class that extends `ContentTransferService` and acts as a controller for the UCF processor and transport classes as well as the WDK container class. The service sets up the DFC context (DFC session, operation) and reads the WDK container definition. Each content transfer container has a service implementation.

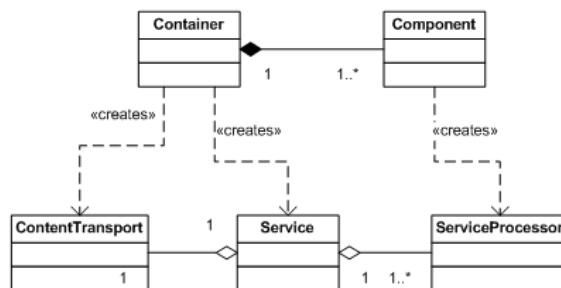
The container definition also specifies a transport class that encapsulates logic for the transport mechanism. The class `HttpContentTransport` implements HTTP transport, and `UcfContentTransport` implements UCF transport. This class interacts with the service class, the underlying DFC operation, WDK content transfer component, and UCF (if applicable). There is only one transport per service instance.

The content transfer component definition specifies a processor class that extends `InboundProcessor` or `OutboundProcessor`. The component processor sets object properties on the DFC package and operation objects. The processor operates independently of the type of transport. Each content transfer component has a processor implementation.

Some containers also use processor classes. For example, the edit container processor class launches the editor applications on the client.

Each content transfer component is paired with a container. The component definition specifies the processor class. The container definition specifies the service and transport classes. The container component is responsible for creating an instance of the service class, setting it up with the desired transport class implementation and invoking the service. The component class creates an instance of the processor class, sets appropriate property values on it, and supplies it to the container, which executes the service class. The relationship between the classes is diagrammed below:

Figure 17-4. Content transfer component classes and service layer



Both service classes and service processor classes provide hooks for pre- and post-processing. These customization points are described in [UCF transfer component customization, page 533](#)

The ContentTransferConfig class provides access to the application content transfer configuration in app.xml, such as transfer mechanism and server and client content locations. A custom configuration reader class may be specified in ContentTransferConfig.properties.

## UCF transfer component customization

You can extend a content transfer component or container. You can also extend or write your own service class for the container or service processor class for the component. provide hooks for pre- and post-processing.

Refer to [Content transfer listeners, page 531](#) for the return values such as new object IDs that are available from the content transfer container classes.

A container service class extends the abstract class ContentTransferService in the package com.documentum.web.contentxfer. For example, the export container service class, com.documentum.web.contentxfer.impl.ExportService, extends OutboundService in the same package, which itself extends ContentTransferService. The following methods of ContentTransferService are good customization points:

- preExecute()
  - Initializes the underlying operation and package objects
- postExecute()
  - Called at the end of execute()

A component processor class extends the abstract class ServiceProcessorSupport and implements IServiceProcesso in the package com.documentum.web.contentxfer. For example, the export component processor class com.documentum.web.contentxfer.impl.

ExportProcessor extends OutboundProcessor, which extends BaseProcessor, which extends ServiceProcessorSupport.

The following methods of ServiceProcessorSupport are good customization points:

- `preProcess(IDfContentPackage pkgp)`  
Called to pre-process the operation package, if one exists, before executing the operation.
- `preProcess(IServiceOperation op)`  
Called to pre-process the service operation before executing it. If the operation package exists, this method is called after `preProcess(IDfContentPackage)`.
- `postProcess(IDfContentPackage pkgp)`  
Called to post-process the operation package, if one exists, after executing the operation.
- `postProcess(IServiceOperation op)`  
Called to post-process the operation after executing it.

## Content transfer control initialization

Any component that extends ContentTransferComponent can provide configuration of named controls in the component definition. For example, you can specify default attribute values on the control.

Controls that can be initialized are specified in the `<init-controls>` element of the component definition. The `<control>` element has an attribute for the control name and type. The name must match the control name in the JSP page, and the type must match the control class. Each attribute value to be initialized is specified within an `<init-property>` element. The attribute name is specified as the value of `<property-name>` and the value as `<property-value>`. For example, the `cancelcheckout` component JSP page `cancelCheckout.jsp` has a radio control named `nodescendents`:

```
<dmf:radio name="nodescendents" group="group1" nlsid="
MSG_LEAVE_DESCENDENTS_CHECKEDOUT"/>
```

This control is initialized in the component definition as follows:

```
<init-controls>
 <control name="nodescendents" type="com.documentum.web.form.control.Radio">
 <init-property>
 <property-name>value</property-name>
 <property-value>>true</property-value>
 </init-property>
 </control>
</init-controls>
```

**Note:** Changes to the configured control's properties by the component implementation class at runtime override the configured defaults.

## Content transfer debugging

Content transfer tracing flags UCF\_MANAGER or HTTP\_MANAGER can be turned on in `com.documentum.debug.TraceProp` properties or by navigating to `/wdk/tracing.jsp`. Output is found in the WDK log file whose location is specified in `$DOCUMENTUM_SHARED/config/log4j.properties`. By default, this file is the value of `log4j.appender.file.File=C\:/Documentum/logs/wdk.log`.

**Note:** The logging is verbose and can generate large logs in a short period of time.

Sample tracing output for UCF checkout operation, from `wdk.log` (edited for appearance):

```
[http-8080-Processor21] DEBUG com.documentum.web.common.Trace-
ucf required: true, component: checkoutcontainer

[http-8080-Processor22] DEBUG com.documentum.web.common.Trace-
ucf required: true, component: ucfinvoker

[http-8080-Processor21] DEBUG com.documentum.web.common.Trace-
Received request key:
2284f8c1b0cfef3b1q113108b1q10338b260041q1d7ffe

[http-8080-Processor21] DEBUG com.documentum.web.common.Trace-
Received client id:
1;2284f8c1b0cfef3b1q113108b1q10338b260041q1d7ffe, httpsession:
E4B2DF134E335B0C03AF2AD13108D098

[http-8080-Processor21]
DEBUG com.documentum.web.common.Trace-
new session id added:
1;2284f8c1b0cfef3b1q113108b1q10338b260041q1d7ffe

[http-8080-Processor21]
DEBUG com.documentum.web.common.Trace-
reserved
2284f8c1b0cfef3b1q113108b1q10338b260041q1d7ffe by
Thread[http-8080-Processor21,5,main]

[http-8080-Processor21] DEBUG com.documentum.web.common.Trace-
ucf required: true, component: checkoutcontainer

[http-8080-Processor21] DEBUG com.documentum.web.common.Trace-
created new session:
com.documentum.ucf.server.transport.impl.ServerSession@1154718, id: 1

[http-8080-Processor21] DEBUG com.documentum.web.common.Trace-
reserved com.documentum.ucf.server.transport.
impl.ServerSession@1154718 by Thread[http-8080-Processor21,5,main]
```

```
2005-04-12 16:21:05,244 351167 [http-8080-Processor21]
DEBUG com.documentum.web.common.Trace -
get new session: 1, refcount: 1

2005-04-12 16:21:05,260 351183 [http-8080-Processor21]
DEBUG com.documentum.web.common.Trace -
unreserved UcfSessionManager$SessionKey
[1;2284f8c1b0cfef3b1q113108b1q10338b260041q1d7ffe] by Thread
[http-8080-Processor21,5,main]

2005-04-12 16:21:11,904 357827 [http-8080-Processor23]
DEBUG com.documentum.web.common.Trace -
INotificationMonitor.notifyProgress
[stage=UCF_I_UPLOAD_PROGRESS, 0% of 64540 bytes]
```

Sample server tracing for HTTP view operation, from wdk.log (edited for appearance):

```
[Thread-34] - add outgoing: content id em89j39g23oa7jj6p,
file D:\Documentum\contentXfer\user2ks-2005.04.01-1712h.58s_8930\
fedfb61q1030077c2e01q1d8000\107_0707.jpg

[Thread-34] - set download event: content id em89j39g23oa7jj6p

[http-8080-Processor3] - sent outgoing:
content id em89j39g23oa7jj6p, file D:\Documentum\contentXfer\
user2ks-2005.04.01-1712h.58s_8930\fedfb61q1030077c2e01q1d8000\
107_0707.jpg, method 2, type image/jpeg

[http-8080-Processor3] - removing outgoing:
content id em89j39g23oa7jj6p
```

For information on UCF logging, refer to [UCF logging, page 523](#).

## Using Pre-5.3 content transfer components

UCF content transfer is new in WDK 5.3. The WDK 5.2.5 content transfer components are still present in the WDK runtime to support your customizations until you migrate them. You cannot address a WDK 5.2.5 content transfer component by URL or jump to it in a component class, because the old and new components have the same name. Components with a 5.2.5 version are not launched by the component dispatcher. When a content transfer component is launched by URL, component nest or jump, or action, the new content transfer components are launched by default.

To use your customized WDK 5.2.5 content transfer component, make sure your content transfer component and container extend the WDK 5.2.5 component and container. For example, your custom import component definition, located in /custom/config, would be similar to the following:

```
<component id="import:webcomponent/config/library/
```



```
importContent/import_component.xml">
...
</component>
```

The import action should launch your component instead of the WDK 5.3 import component. If you wish to extend WDK 5.3 content transfer components, their definitions are located in `/webcomponent/config/library/contenttransfer`, for example:

```
/webcomponent/config/library/contenttransfer/importcontent/import_component.xml
```

## Streaming content to the browser

HTTP content transfer streams web-viewable content to the browser. If the application server is running UCF content transfer, additional modes of content streaming are available:

- All content that the user has set a preference for viewing in the browser will be streamed to the browser using HTTP content transfer
- You can force streaming using the `wdk5-download` servlet if the object ID is known

The following example is a URL to the streaming servlet:

```
http://localhost:8080/webtop/wdk-download?objectId=0900000180279b00
```

The save vs. open behavior for streamed content is set in the browser. The user can also right-click on the `wdk-download` link and save the file. The `wdk-download` servlet requires a WDK5 session. For links in non-WDK pages that do not have a session, use the `getcontent` component, which will bring up a login dialog. For example:

```
/webtop/component/getcontent?objectId=0900000180279b00
```

## Content transfer progress

To display progress during content transfer, add a progress listener using the `addProgressListener()` method of `ContentTransferService`. Pass in a listener class that implements `IServiceProgressListener`. Progress is reported at the completion of each `IDfOperationStep` and at the end of the operation. For example, the `JobStatus` class registers a progress listener. The `progressChanged()` implementation adds to the status report at the end of each step. The `progressEnded()` implementation finishes the status report:

```
public void progressEnded(ServiceProgressEvent e)
{
 setStatusReport(new StatusReportEx(
 this, null, null, StatusState.JOB_FINISHED, 100, null));
}
```

}

## Customizing Authentication

Authentication is performed by the authentication service, which tries all of the authentication schemes that are configured for the application until the user is successfully authenticated. You can implement your own authentication scheme that uses your policy servers or business processes.

Authentication and sessions are described in the following topics:

- [Authentication service, page 539](#)
- [Authentication schemes, page 540](#)
- [Silent login, page 542](#)

For information on configuring authentication and login, refer to [Application login and authentication, page 103](#).

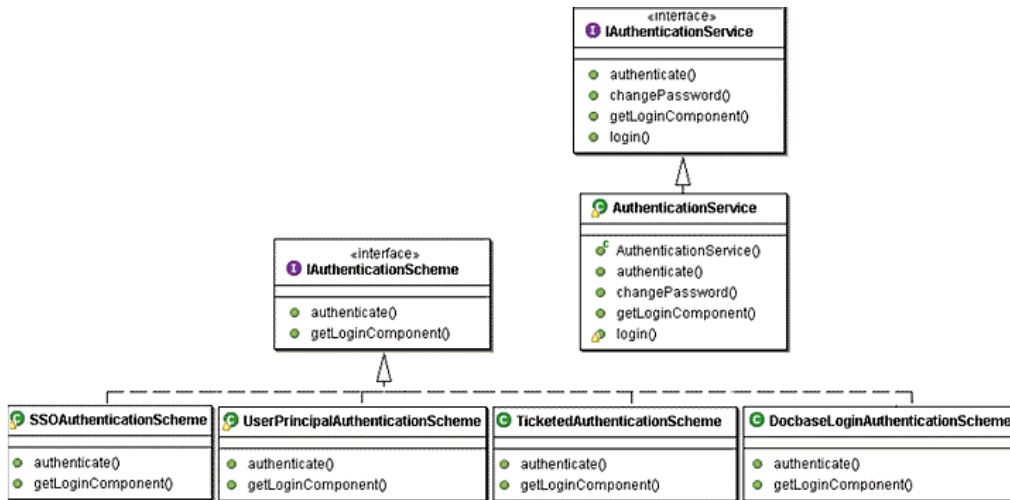
### Authentication service

The authentication service authenticates users with an authentication scheme. The class `AuthenticationService` provides the default implementation of the authentication service interface `IAuthenticationService`. The implementation class also encapsulates the pluggable authentication scheme framework.

You can provide your own implementation of `IAuthenticationService` by specifying the class in the element `<authentication>.<service_class>` element in `app.xml`. One of the main uses of a custom authentication service is for an application to pre- or post-process the authentication service. For example, the Web Publisher authentication service extends `AuthenticationService` and overrides `authenticate()` to test the user domain.

The authentication service interfaces and schemes are diagrammed below:

Figure 18-1. Authentication service interfaces



## Authentication schemes

The authentication service uses a list of authentication schemes to perform authentication. The list is defined in `com.documentum.web.formext.session.AuthenticationSchemes` properties. The service tries each scheme in the order listed in the properties file to authenticate a user.

The authentication service processes registered authentication schemes in the order that they appear in the properties file. The schemes must be indexed sequentially.

The types of schemes supported by WDK authentication are described in [Application login and authentication, page 103](#). The scheme implementation classes are the following:

- `TicketedAuthenticationScheme`
- `SSOAuthenticationScheme` (Single sign-on)
- `UserPrincipalAuthenticationScheme`
- `SavedCredentialsAuthenticationScheme` (available only in WDK for Portlets)
- `SingleDocbasePerDocbrokerUserPrincipalAuthenticationScheme` (available only in WDK for Portlets)
- `DocbaseLoginAuthenticationScheme` (per-session authentication)

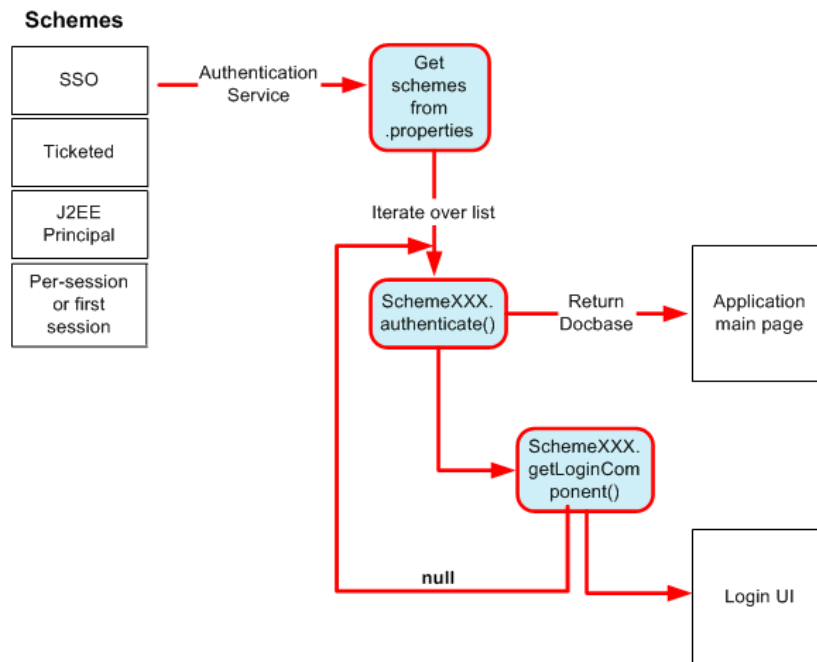
Presents a login dialog. After a successful Content Server login, the scheme will store the password in the portal preference store.



**Caution:** The DFC session manager does not support mixed authentication schemes. For example, if the user logs into the first repository with a user principal login and logs into the second repository with a session login dialog, the second login attempt will throw an exception.

The authentication service will iterate the list of schemes in the order that they are specified in the properties file. The service will invoke the scheme's `authenticate()` and `getLoginComponent()` methods and will stop the process when a scheme method returns a valid string. This sequence is diagrammed below:

**Figure 18-2. Authentication scheme processing**



An authentication scheme is an instance of a class that implements the interface `IAuthenticationScheme`. This interface defines two methods:

- `authenticate(HttpServletRequest request, HttpServletResponse response, String docbase)`

Authenticates a user based on the current HTTP request. The method returns the repository name in which the user was authenticated. If `null` is returned, the authentication has failed. The `HttpServletRequest` parameter can be any information from the request, such as a header or cookie. The repository name parameter is optional, and the `authenticate` method can obtain the repository name from another source.

- `getLoginComponent(HttpServletRequest request, HttpServletResponse response, String doctype, ArgumentList args)`

If authentication fails, the authentication service calls `getLoginComponent()` and launches the specified component. The arguments are the same as those for `authenticate`, with the addition of the `ArgumentList`, which can be populated by the authentication scheme class with arguments that are passed to the login component.

**Note:** Your custom authentication scheme must be registered as the first authentication plugin in the list in `com.documentum.web.formext.session.AuthenticationSchemes` properties.

## Silent login

Several forms of silent login are supported in WDK:

**External resource login** — In the `onInit()` method, get the login details from an outside source such as a property file or LDAP. You should display an error message in your derived class for login failure. The following example shows how to implement login from a properties file.

### Example 18-1. Login from external resource

Add login properties of username, password, and domain to a properties file. For example, create a file in `/WEB-INF/classes/com/mycompany/session` named `SilentAuthentication.properties` with the following content:

```
#DOCBASE_NAME=USERNAME, PASSWORD, DOMAIN
testDocbase=testUser2,pwdXXX
```

Create an authentication scheme that implements `IAuthenticationScheme` and reads the login properties, similar to the following class:

```
package com.mycompany.session;
import java.io.*;
import java.util.*;
import javax.servlet.http.*;
import com.documentum.fc.common.*;
import com.documentum.web.common.ArgumentList;
import com.documentum.web.formext.session.*;

public class SilentAuthenticationScheme implements IAuthenticationScheme
{
 /**
 * Properties filename that contains the defined credentials.
 */
 private static String RES_BUNDLE_NAME = "SilentAuthentication.properties";

 /**
 * Resource bundle that loads the credentials.
 */
}
```

```

private static PropertyResourceBundle s_resBundle = null;

/**
 * Implements IAuthenticationScheme#authenticate(...)
 * by reading credentials from the SilentAuthentication.properties file
 * and creating sessions for the credentials.
 * @param request HTTP request.
 * @param response The HTTP response. Not used in this method.
 * @param docbase Docbase name. Ignored in this implementation.
 * @return docbase The docbase into which the user was logged in.
 * For multiple credentials, the first docbase name in
 * SilentAuthentication.properties is returned.
 */
public String authenticate(HttpServletRequest request,
 HttpServletResponse response, String docbase) throws DfException
{
 IAuthenticationService service = AuthenticationService.getService();
 HttpSession sess = request.getSession();
 initCredentials();

 Enumeration docbases = s_resBundle.getKeys();
 String defaultDocbase = null;
 while (docbases.hasMoreElements())
 {
 String docbaseName = (String) docbases.nextElement();
 String value = s_resBundle.getString(docbaseName);

 ArrayList creds = parseCredentials(value);
 String username = (String) creds.get(0);
 String password = (String) creds.get(1);

 String domain = null;
 if (creds.size() > 2)
 {
 domain = (String) creds.get(2);
 }

 service.login(sess, docbaseName, username, password, domain);

 if (defaultDocbase == null)
 {
 defaultDocbase = docbaseName;
 }
 }
 return defaultDocbase;
}

/**
 * Implements IAuthenticationScheme#getLoginComponent(...).
 * Return null because silent authentication does not have a login component
 * @return null
 */
public String getLoginComponent(HttpServletRequest request, HttpServletResponse
 response, String docbase, ArgumentList outArgs)
{
 return null;
}

```

```
/**
 * Loads the properties file that contains credentials
 */
protected void initCredentials()
{
 try
 {
 if (s_resBundle == null)
 {
 InputStream is =
 SilentAuthenticationScheme.class.getResourceAsStream(
 RES_BUNDLE_NAME);
 s_resBundle = new PropertyResourceBundle(is);
 }
 }
 catch (IOException ioe)
 {
 //error handling
 }
}

/**
 * Parses a string "username,password,domain" into an ArrayList object.
 * @param creds ArrayList of credentials.
 * At index: 0=>username 1=>password 2=>domain
 * @return
 */
protected ArrayList parseCredentials(String creds)
{
 ArrayList listCreds = new ArrayList(3);
 StringTokenizer tokCreds = new StringTokenizer(creds, ",");
 int cntr = 0;
 while (tokCreds.hasMoreTokens())
 {
 String cred = tokCreds.nextToken();
 listCreds.add(cntr, cred);
 cntr++;
 }
 return listCreds;
}
}
```



## Managing Sessions

Documentum sessions are acquired and managed through `IDfSession`, for components and actions, or through `IDfSessionManager`, for other classes.

The HTTP session objects are available as JSP implicit objects in both the JSP page and in servlets. The `SessionState` class encapsulates HTTP session context.

The `ClientSessionState` class encapsulates the client browser session state. This class is used by `AppSessionContext` to restore the client's selected repository when a browser is refreshed. It is also used by the API that handles the client repository. For client environments such as Application Connectors or portlets (clientenv setting in `app.xml`), this functionality is turned off and `ClientSessionState` delegates calls to `SessionState`.

Session management is described in the following topics:

- [Getting a session in a component or action class, page 545](#)
- [Getting a session using `SessionManagerHttpBinding`, page 547](#)
- [Storing and retrieving objects in the session, page 549](#)
- [Binding and caching in a request thread, page 550](#)
- [Application, session, and request listeners, page 550](#)
- [IDfSessionManagerEventListener, page 551](#)
- [Session synchronization, page 552](#)
- [Session tracing, page 552](#)
- [JSP implicit objects in WDK, page 553](#)

## Getting a session in a component or action class

Components acquire a session by the methods `getDfSession()` or `getDfSession(REQUEST_LIFETIME` or `COMPONENT_LIFETIME)`. For example:

```
import com.documentum.fc.client.IDfSession;
```

```
...
IDfSession dfSession = getDfSession();
```

REQUEST\_LIFETIME specifies that the session is held until the end of the HTTP request. Note that getDfSession() calls getDfSession(REQUEST\_LIFETIME). If you call getDfSession() more than once within the same request, the same session will be returned.

COMPONENT\_LIFETIME specifies that the session is held until the component exits or the HTTP session times out, whichever occurs first. For performance and scalability, you should not use COMPONENT\_LIFETIME unless necessary. Use request lifetime so that sessions are held only briefly.

**Note:** Use REQUEST\_LIFETIME to ensure session cleanup. There is no guarantee that a session will be released if it is stored for longer than a request lifetime. For example, a user may close the browser.

The Component class method that acquires a session, getDfSession(), obtains the session through SessionManagerHttpBinding. The session is released when the component is destroyed.



**Caution:** Do not store IDfSession objects as member variables. The session may time out and cause a runtime error. Instead, every time a session is needed in that class, call the Component class getDfSession() method. IDfTypedObjects obtained through IDfCollection do not cause a problem. (They are a memory-cached row from a collection.)

#### **Example 19-1. Getting a session in a component class**

In the following example from the component class DeleteQueueItem, the deleteItem() method gets a session in order to perform the repository operation:

```
private boolean deleteItem()
{
 boolean retval = false;
 try
 {
 ...
 // perform dequeue
 IDfSession dfSession = getDfSession();
 dfSession.dequeue(new DfId(m_strObjectId));
 // error check here
 retval = true;
 }
 ...
}
```

An action class can get a session by calling getDfSession on the component instance that is passed into execute() and queryExecute().

**Example 19-2. Getting a session in an action class**

In the following example from the action class `AddComponentFSPrecondition`, the `queryExecute()` method gets a session using the component instance that is passed in by the action service:

```
public boolean queryExecute(String strAction, IConfigElement config,
 ArgumentList arg, Context context, Component component)
{
 boolean bExecute = false;
 IDfSession dfSession = component.getDfSession();
 try
 { ... }
}
```

**Note:** The `IDfSessionManager` interface methods `getSession()` and `release()` can be nested. That is, if `getSession()` is called twice for the same repository, the same `IDfSession` object will be returned each time. If `release()` is called twice on that `IDfSession` object, the session will not be released until the second call. This nesting behavior ensures that the same session is used for all `DocbaseObject` and `DocbaseAttribute` control actions, for example.

## Getting a session using `SessionManagerHttpBinding`

You can get a Documentum session in any class or JSP page using the static methods of `SessionManagerHttpBinding`. You must explicitly release a session that is obtained with `SessionManagerHttpBinding`. If you get a session within a component class using `getDfSession()`, the session is automatically released at the end of the component lifetime.

To access a session from a non-component class, call the helper class `SessionManagerHttpBinding` to return the `IDfSessionManager` instance and the current repository name. `IDfSessionManager` binds to the HTTP sessions and can store and retrieve the current session. You can also get and set the current repository.

To get a session, use the following syntax:

```
IDfSessionManager sessionManager =
 SessionManagerHttpBinding.getSessionManager();
IDfSession dfSession = null;
try
{
 dfSession = sessionManager.getSession(strDocbase);
 ...
}
```

**Note:** `getSession()` will throw an exception if the user's identity has not been set for the current repository and the user has not been authenticated.

You must release all sessions that you obtain through `IDfSessionManager`. Release the session in the request that acquired it. It is recommended that you also release the session in a finally block. For example:

```
finally
{
 if (dfSession != null)
 {
 sessionManager.release(dfSession);
 }
}
```

After you release the session, the session is kept alive by the session manager for 5 seconds when connection pooling is enabled. If the session is not reclaimed, the session is disconnected. This allows a client to set the session shortly after releasing it without a performance penalty. If connection pooling is disabled and the session is released, the session timeout occurs after several minutes.

### **Example 19-3. Getting a session with `SessionManagerHttpBinding`**

Following is an example of a component class that gets and releases a session and gets the current repository:

```
public class MyComponent extends Component
{
 /**
 * Handle an action event
 * @param c the control that raised the event
 * @param a the event arguments
 */
 public void onActionEvent(Control c, ArgumentList a)
 {
 //get the session Manager
 IDfSessionManager sessionManager =
 SessionManagerHttpBindingBinding.getSessionManager();
 IDfSession dfSession = null;
 try
 {
 //get the current Docbase name and session
 dfSession = sessionManager.getSession(
 SessionManagerHttpBindingBinding.getCurrentDocbase());
 ...
 }
 ...
 finally
 {
 if (dfSession != null)
 {
 sessionManager.release(dfSession);
 }
 }
 }
}
```

## Storing and retrieving objects in the session

WDK session state is "browser aware": WDK will store and retrieve the navigation location and other state information on a per browser window basis.

Session state is managed by `com.documentum.web.common.SessionState`. You can store and retrieve some objects in the session by calling `setAttribute(String strAttrName, Object oValue)`. Do not store session variables using `HttpSessionState` or `HttpSession`.

**Note:** Do not store objects in the session if your component implements `Serialized` unless they are marked as `transient`. DFC and BOF objects cannot be serialized.

To support multiple browser windows, add a `__dmfClientId` request parameter on URLs. This parameter will be used to maintain session state. When you construct a URL in JavaScript, call `addBrowserIdToURL()`. For example:

```
var url = addBrowserIdToURL("<%=strUrl%>");
location.replace(url);
```

If a WDK-based application does not store session variables using `SessionState` and does not create URLs using `addBrowserIdToURL`, session state will be shared by multiple browser windows, as in previous versions of WDK.

### Example 19-4. Storing and retrieving a string in `SessionState`

In the following example from `ObjectLocator`, the user's previous navigation path is stored:

```
protected void setPreviousContainerNavigated(String strPath)
{
 String strVarName = getDocbaseSessionVariableKey(
 CONTAINERPATH_SESSION_VAR+'\n' + getInitialDocbaseType());
 if (strPath == null || strPath.length() == 0)
 {
 SessionState.removeAttribute(strVarName);
 }
 else
 {
 SessionState.setAttribute(strVarName, strPath);
 }
}
```

The value is retrieved with `getAttribute()`:

```
protected String getPreviousContainerNavigated()
{
 // get initial folder path
 String strContainerPath = (String)SessionState.getAttribute(
 getDocbaseSessionVariableKey(
 CONTAINERPATH_SESSION_VAR+'\n' + getInitialDocbaseType()));

 return strContainerPath;
}
```

## Binding and caching in a request thread

The utility class `com.documentum.web.common.ThreadLocalVariable` allows you to bind a variable to a single thread. The class has methods to set and get the thread value and to render the variable to a `String`. The `finalize()` method removes the variable from the dictionary of known variables.

The class `com.documentum.web.common.ThreadLocalCache` class stores and retrieves objects in the scope of the current thread. For example, `ObjectCacheUtil` and `UserCacheUtil` use `ThreadLocalCache` to cache sysobjects, user objects, and user privileges. If you find that a component is getting the same object multiple times, use one of the cache utility classes to cache the object. For example, the `FreezeAssemblyAction` class caches the object to avoid multiple fetches:

```
IDfSession dfSession = component.getDfSession();
String objectIdArg = arg.get("objectId");
IDfSysObject sysobject =
(IDfSysObject) ObjectCacheUtil.getObject(dfSession, objectIdArg);
 if (sysobject.getHasFrozenAssembly() == false)
 {
 if (sysobject.getLockOwner().length() == 0)
 {
 assembledFromId = sysobject.getAssembledFromId();
 if (isAssembly(assembledFromId))
 {
 canExecute = true;
 }
 }
 }
}
```

The class `com.documentum.web.formext.control.docbase.DocbaseAttributeCache` caches `IDfTypedObject` lookups from the data dictionary, which occurs several times per attribute. Objects can be retrieved similar to the following:

```
DocbaseObject obj = (
 DocbaseObject) getForm().getControl(strObject);
IDfTypedObject type = DocbaseAttributeCache.getDfTypedObject(
 strAttribute, obj);
```

In the same package, `DocbaseObjectCache` caches the corresponding `IDfValidator` and `IDfPersistentObject` for each repository object.

## Application, session, and request listeners

The `com.documentum.web.env` package provides the following listener interfaces:

- Application listeners

Listener classes that implement `IApplicationListener` are notified at application start and end. The listener implement must be registered in the `app.xml` element `<listeners>.<application-listeners>`.

- Session listeners

Listener classes that implement `ISessionListener` are notified each time a session is created or destroyed. The listener implement must be registered in the `app.xml` element `<listeners>.<session-listeners>`.

- Request listeners

Listener classes that implement `IRequestListener` are notified at request start and end. The listener implement must be registered in the `app.xml` element `<listeners>.<request-listeners>`.

A listener class for the DFC session manager can be registered in the `app.xml` file as the value of the element `<application>.<dfsessionmanagereventlistener><class>`.

These application, session, and request listeners must be registered in `app.xml` in order to be notified. They are notified by the `WDKController` filter class, which is mapped to all requests (`"/`). For more information on the controller, refer to [Table 2–35, page 85](#).

## IDfSessionManagerEventListener

You can register a listener class for the DFC session manager by adding an entry to your custom `app.xml` file: Add the element `<dfsessionmanagereventlistener>` under `<application>`. Add a child element `<class>` and provide the fully qualified class name of your listener class.

The listener class must implement the DFC interface `com.documentum.fc.client.IDfSessionManagerEventListener`, which defines two methods:

- `onSessionCreate`
- `onSessionDestroy`

### Example 19-5. Implementing a Session Event Listener

In the following example from the Web Publisher listener `WcmSessionManagerEventListener`, the listener performs business logic in `onSessionCreate()`:

```
public void onSessionCreate (IDfSession session) throws DfException
{
 String value = session.apiGet("get", "sessionconfig,_is_restricted_session");
 if (DfUtil.toBoolean(value) == false)
 session.getSessionConfig().appendString("application_code",
 IWcmConstant.APPLICATION_CODE);
}
```

```
public void onSessionDestroy (IDfSession session) throws DfException
{
 //Do nothing
}
```

## Session synchronization

HTTP session synchronization (locking and unlocking) is managed through `com.documentum.web.common.SessionSync`. The component dispatcher, form processor, and control tag all use session locking. For example, `ControlTag` locks the session for `doStartTag()` and `doEndTag()`.

Session locking has a negative impact on performance. If you lock the session, you must unlock in the `Finally` block.

### Example 19-6. Synchronizing the HTTP session

The following example from `Control.doStartTag()` locks and unlocks the session:

```
final public int doStartTag() throws JspTagException
{
 try
 {
 SessionSync.lock(pageContext.getSession());
 m_bInRenderStart = true;
 renderStart(pageContext.getOut());
 }
 ...
 finally
 {
 m_bInRenderStart = false;
 SessionSync.unlock(pageContext.getSession());
 }
}
```

## Session tracing

By default, a single session is traced when tracing is enabled. To enable tracing for the current session, visit `/wdk/tracing.jsp` tool and check the box that enables tracing for the current HTTP session.

To trace all sessions, set `SESSIONENABLEDBYDEFAULT` to `true` using `tracing.jsp` or by editing `WEB-INF/com/documentum/debug/Trace.properties`.



## JSP implicit objects in WDK

The following JSP objects are used extensively by the WDK libraries. Objects can be used in JSP pages or in Java classes and can be referenced in the JSP page or Java class by their JSP object name. Each object is referenced here by its JSP object name, with the class name in parentheses:

**pageContext (javax.servlet.jsp.PageContext)** — Context for the current page. Variables that are scoped to the page context are available while the page is being processed. The pageContext object is available to handlers in your Java classes in order to output JavaScript dynamically to the browser.

### Example 19-7. Printing JavaScript dynamically to the browser

Use the PageContext object to get the current form and output to it. The following example from the Web Publisher class WpStatusBar writes JavaScript to the browser in its implementation of IRequestListener.notifyFinish():

```
public void notifyFinish(PageContext pageContext)
{
 //Get the top level form here
 Form form = (Form)pageContext.getAttribute(
 FORM, PageContext.REQUEST_SCOPE);

 WpAsyncTaskListener listener =
 ...
 String value = WpStatusBar.getClientMessageId(listener);
 StringBuffer buf = new StringBuffer(1024);
 // Output initialisation function (called using timeout)
 buf.append("<script>function _updateTaskStatus() {\n");
 buf.append("fireClientEvent('")
 .append(UPDATE_TASK_STATUS_EVENT)
 .append("'", "'")
 .append(UPDATE_TASK_STATUS_FUNCTION)
 .append("'", "'")
 .append(value)
 .append("'", self.name);
 buf.append("</script>\n");
 // Invoke initialisation function on timer
 buf.append("<script>setTimeout('_updateTaskStatus()', 50);</script>\n");
 ...
 pageContext.getOut().print(buf);
 //error handling
}
```

**request (javax.servlet.http.HttpServletRequest)** — The HTTP request or URL, which is sent by the client browser to the server. You can access request parameters, attributes, headers, and cookies.

### Example 19-8. Accessing request parameters

The following example from repeatingAttributes.jsp passes in the request object to create a URL to a JavaScript page:

```
<script src='<%=Form.makeUrl(request, "/wdk/include/modal.js")%>'
 language='JavaScript1.2'>
```

**response (javax.servlet.http.HttpServletResponse)** — HTTP response, or the HTML that is sent from the JSP container to the client browser. You can access the response object in your Java class to set cookies or to print out the response.

**Example 19-9. Writing a response from a servlet**

In the following example from `WorkflowEditorServlet`, the response object is used to write output from the servlet:

```
responseResult = handleResourceRequest(
 session, (IRequest) requestObject, locale);
// Send response.
response.setContentType(Constants.DEFLATED_JAVA_SERIALIZED_CONTENT);
DeflaterOutputStream deflaterOut = new DeflaterOutputStream(
 response.getOutputStream());
out = new ObjectOutputStream(deflaterOut);
out.writeObject(responseResult);
deflaterOut.finish();
out.flush();
```

**out (javax.servlet.jsp.JspWriter)** — Prints to the response message body. This object is used extensively by the tag classes.

**Example 19-10. Writing HTML output in a tag class**

In the following example from the tag class `HiddenTag`, the JSP writer is used to write HTML output:

```
protected void renderEnd(JspWriter out)
 throws IOException
{
 Hidden hidden = (Hidden) getControl();
 StringBuffer buf = new StringBuffer(256);
 buf.append("<input type='hidden' ").append(renderNameAndId());
 buf.append(" value='")
 .append(formatText(hidden.getValue()))
 .append(">");
 out.println(buf.toString());
}
```

**Note:** URLs in JSP pages must have paths relative to the Web application root context or relative to the current directory. For example, the included file `<%@ include file='doclist_thumbnail_body.jsp' %>` is in the same directory as the including file. The included file `<%@ include file='/webcomponent/navigation/drilldown/drilldown_body.jsp' %>` is in the `/webcomponent` subdirectory of the Web application.

Refer to third-party JSP documentation for more information on how to use JSP implicit objects.



## Customizing Search

The following topics describe commonly used features in search customization:

- [Programmatic search value assistance, page 557](#)
- [Troubleshooting search, page 558](#)
- [Search class diagrams, page 560](#)

For information on default search behavior and configuration options, refer to [Configuring search, page 144](#).

### Programmatic search value assistance

Data dictionary value assistance is available in 5.3 advanced search, unless the value assistance is conditional. If you have not defined value assistance for an attribute in the repository data dictionary, you can add value assistance programmatically. You must define a custom tag handler to render the value assistance values. The tag handler is specified in the search configuration file `advsearchex.xml` as follows:

```
<searchvalueassistance>
 <attribute_type_name>
 fully_qualified_class_name
 </attribute_type_name>
</searchvalueassistance>
```

When the user selects an attribute for search, the values in the criteria dropdownlist control will be populated by the custom tag class. To add your own custom tag class, copy the file `/wdk/advsearchex.xml` to `/custom/config` and add your handlers to the `<searchvalueassistance>` element. Your tag handler must implement `ISearchAttributeValueTag`.

**Note:** Do not delete the Documentum value assistance handlers, because the entire contents of the `<searchvalueassistance>` will override the contents of the element in the WDK version of this file.

The following tag handlers render values for certain attributes. The handler classes are in `com.documentum.web.formext.control.docbase.search`.

- **BooleanVATag**  
Provides values for any Boolean attribute
- **ContentTypeVATag**  
Provides valid `a_content_type` (`dm_format`) names and descriptions
- **ExistingValueVATag**  
Uncomment this tag and specify an attribute for which to populate the dropdown list with all existing values for the selected object type
- **ObjectTypeVATag**  
Populates the search object type dropdown list with available object types
- **PermissionVATag**  
Provides possible permission values (none, browse, read, relate, version, write, delete) for setting `world_permit`, `group_permit`, and `owner_permit` attributes
- **SearchMetaDataVATag**  
Gets attribute names, default value, and description for each attribute. This handler is for internal use only.

Your tag class should extend the abstract class `SearchVADropDownListTag` and implement `ISearchAttributeValueTag`. For example, the `BooleanVATag` class implements `populateValueDropDownList` to provide the two boolean values:

```
protected void populateValueDropDownList(SearchDropDownList ddList)
{
 Option optionTrue = new Option();
 optionTrue.setValue("1");
 optionTrue.setLabel(SearchControl.getString("MSG_TRUE", ddList));
 ddList.addOption(optionTrue);

 Option optionFalse = new Option();
 optionFalse.setValue("0");
 optionFalse.setLabel(SearchControl.getString("MSG_FALSE", ddList));
 ddList.addOption(optionFalse);
}
```

## Troubleshooting search

You can enable query debugging in `$DOCUMENTUM_HOME/config/log4j.properties`:

- Debug DQL queries by adding the following line:  
`log4j.logger.com.documentum.fc.client.search.impl.broker.DocbaseBroker=DEBUG`

- Debug ECI queries by adding the following line:

```
log4j.logger.com.documentum.fc.client.search.impl.broker.ECISBroker=DEBUG
```

After restarting the application, exercise your query and examine the log. The log file name and location is specified in the `log4j.properties` file. The following excerpt from `wdk.log` traces a query against a 5.2.5 repository:

```
2005-04-20 15:04:20,300 362317 [DocbaseBroker:processing]
DEBUG com.documentum.fc.client.search.impl.broker.DocbaseBroker -
InternalProcess dql=SELECT r_object_id,object_name,r_object_type,r_lock_owner,
owner_name,r_link_cnt,r_is_virtual_doc,r_content_size,a_content_type,
i_is_reference,r_assembled_from_id,r_has_frzn_assembly,a_compound_architecture,
i_is_replica,r_modify_date FROM dm_sysobject WHERE (
object_name LIKE %5.3% ESCAPE \) AND FOLDER(
/Product Info/ 5.3 Products/ECI 5.3/Engineering/Functional Specs,DESCEND) AND (
a_is_hidden = FALSE)
```

```
2005-04-20 15:04:20,721 362738 [DocbaseBroker:processing]
DEBUG com.documentum.fc.client.search.impl.broker.DocbaseBroker -
Send 1 results for action query: com.documentum.fc.client.search.impl.
DfQueryBuilder@9801f4 sources:dm_notes
```

**Document not returned by search** — Check the following list of possible causes:

- Check whether the docbase is indexed
- Verify that Indexing Agent is running on the Content Server and that the expected document has been properly indexed:
  1. Connect to `http://localhost:15100` on the host where Index Server is running.
  2. In the query box, enter: `dmftkey:id_of_object`, substituting the object ID for the object to be searched for.
  3. Click **Search**.

The XML data displayed on the screen should have the proper object metadata. If you see object metadata, then the reference object has been indexed.

- Verify that the DQL query contains `ENABLE(FTDQL)`

**Performance issues** — The following tips may speed up search performance:

- Remove the display of folder location in `search_component.xml`.
- Configure case-insensitive search on pre-5.3 or non-indexed 5.3 docbases
- Use FTDQL hint to optimize DQL queries. Can be used to add database-specific hints or turn off search for indexed properties. DQL hints are documented in *Content Server DQL Reference Manual*.
- Check bandwidth between the application server and Content server

**Limitations** — The following limitations may affect your expected search results:

- Basic search:
  - No search on content-less objects such as folders on a 5.2.5 repositories

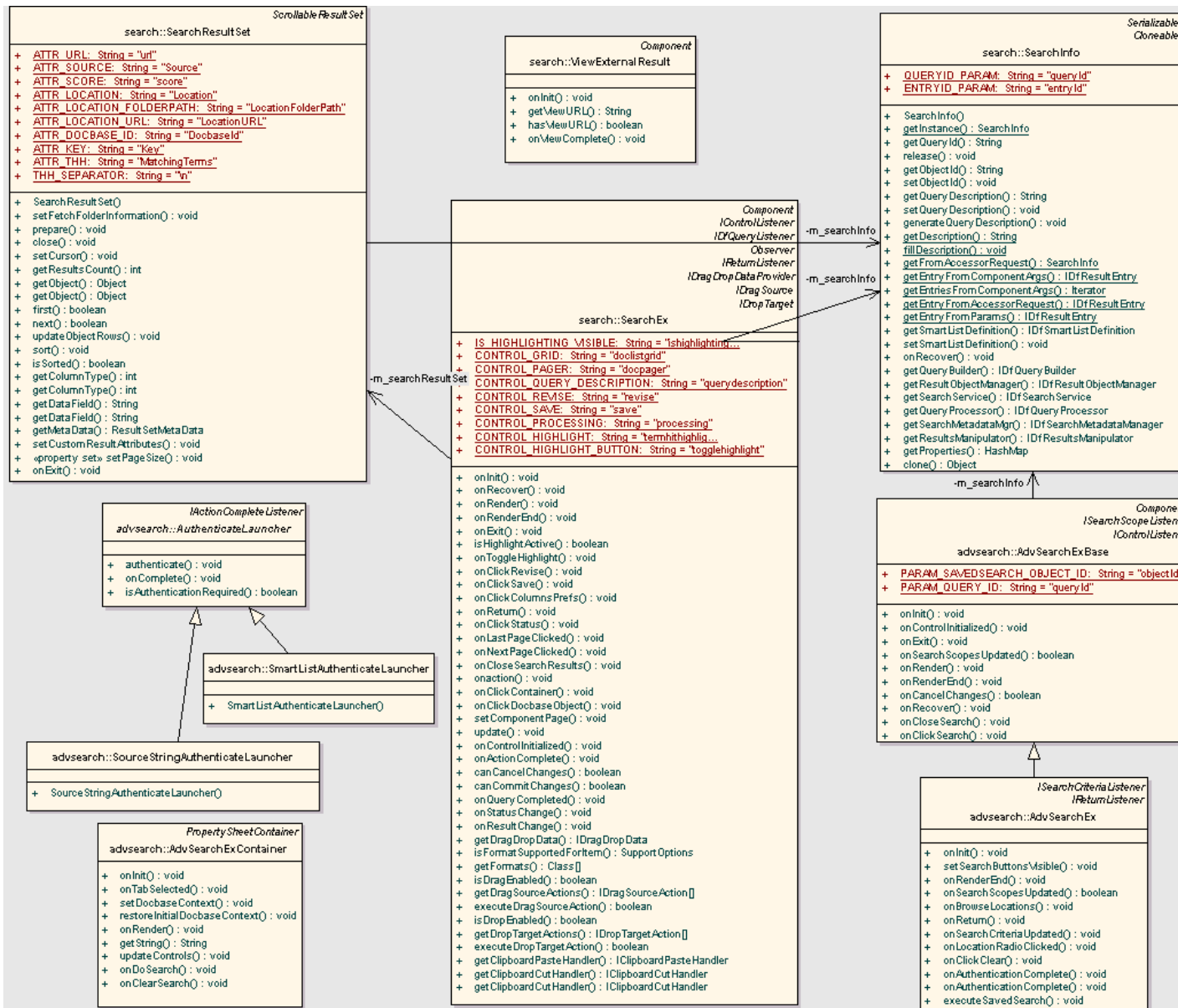
- No search on attributes in 5.2.5 repositories (must use advanced search)
- No wild-card matching
- Failover is not supported
- Advanced search:
  - No explicit Verity support, for example, for Verity semantics such as "does not contain" for pre-5.3 repositories
  - No support for conditional value assistance
  - Property search is not case-sensitive against a 5.3 indexed repository
  - Failover is not supported

## Search class diagrams

The following UML diagram illustrates the search component classes and their dependencies:



Figure 20-1. Search component UML diagram





## Implementing Component and User Preferences

Component-level preferences are defined in the component definition file and implemented in the component behavior class. They can be exposed through the component UI in order to override the component preference on a per-user basis (for example, the login showoptions preference), but most components do not expose the component preferences.

The preferences service adds support for persistent user preferences. The preferences service gets and sets user preferences using a persistent store class that writes cookies to the client machine.

Preferences are described in the following topics:

- [Creating a component preference, page 563](#)
- [Storing and retrieving component preferences, page 566](#)
- [Storing and retrieving user preferences, page 567](#)
- [Tracing preferences, page 569](#)

### Creating a component preference

If a component preference does not need to be exposed as a preference for individual users, it is easier to create a custom element whose value is read in your component class. In the following example, a simple class that extends ObjectGrid displays all objects of a certain type. The component definition has a custom element `<showwebviewableonly>`. If set to true, the query that populates the grid gets objects with a specific attribute value of true. The definition is shown below:

```
<component id="webdocs">
 ...
 <columns>
 <column>
 <attribute>object_name</attribute>
 <label><nlsid>MSG_NAME</nlsid></label>
 <visible>true</visible>
 </column>
```

```

...
<column>
 <attribute>tp_edition</attribute>
 <label><nlsid>MSG_EDITION</nlsid>nlsid</label>
 <visible>true</visible>
</column>
<column>
 <attribute>tp_web_viewable</attribute>
 <label><nlsid>MSG_PUBLISH</nlsid></label>
 <visible>true</visible>
</column>
</columns>

<showwebviewableonly>true</showwebviewableonly>
</component>
</scope>
</config>

```

The component class that reads this custom element is shown below. The lines that read and use the custom element value are highlighted:

```

package com.mycompany.webdocs;
import com.documentum.web.common.ArgumentList;
import com.documentum.web.formext.config.ConfigService;
import com.documentum.web.formext.config.Context;
import com.documentum.web.formext.config.IConfigLookup;
import com.documentum.webcomponent.navigation.objectgrid.ObjectGrid;

public class WebDocs extends ObjectGrid
{
 public void onInit(ArgumentList args)
 {
 this.readPrefs();
 super.onInit(args);
 }

 protected String getQuery(String strVisibleAttrs, ArgumentList args)
 {
 StringBuffer strQueryBuf = new StringBuffer(512);
 if (m_bWebViewableOnly==true)
 {
 strQueryBuf.append("SELECT DISTINCT r_object_id,")
 .append(strVisibleAttrs).append(INTERNAL_ATTRS)
 .append(" FROM technical_publications_web WHERE
 tp_web_viewable = true ORDER BY object_name");
 }
 else
 {
 strQueryBuf.append("SELECT DISTINCT r_object_id,")
 .append(strVisibleAttrs).append(INTERNAL_ATTRS)
 .append(" FROM technical_publications_web ORDER BY object_name");
 }
 return strQueryBuf.toString();
 }

 private void readPrefs()
 {
 Boolean bWebViewableOnly = this.lookupBoolean("


```






```

 showwebviewableonly");
 if (bWebViewableOnly != null)
 {
 bWebViewableOnly.toString());
 m_bWebViewableOnly = true;
 }
}
private boolean m_bWebViewableOnly = false;
}

```



The object grid output before the preference is added is shown below. Objects with webviewable set to either true or false are obtained by the query:

**Figure 21-1. Component without preference**

Items per page: 10		Page 2 of 10+	
Sort by: Name   Format   Modified   Lock Owner   Edition   OK to Publish			
	<a href="#">352intunixfcs.pdf</a> Format: pdf Modified: 1/9/02 1:57 PM Edition: Not Applicable OK to Publish: True	<a href="#">Checkout</a> <a href="#">Edit</a> <a href="#">Checkin</a> <a href="#">Cancel Checkout</a> <a href="#">Add To Clipboard</a> <a href="#">More...</a>	
	<a href="#">404c.pdf</a> Format: pdf Modified: 8/25/00 2:07 PM Edition: OK to Publish: False	<a href="#">Checkout</a> <a href="#">Edit</a> <a href="#">Checkin</a> <a href="#">Cancel Checkout</a> <a href="#">Add To Clipboard</a> <a href="#">More...</a>	
	<a href="#">404c.pdf</a> Format: pdf Modified: 10/4/00 3:23 PM Edition: OK to Publish: False	<a href="#">Checkout</a> <a href="#">Edit</a> <a href="#">Checkin</a> <a href="#">Cancel Checkout</a> <a href="#">Add To Clipboard</a> <a href="#">More...</a>	
	<a href="#">411a.pdf</a> Format: pdf Modified: 6/8/01 12:27 PM Edition: OK to Publish: False	<a href="#">Checkout</a> <a href="#">Edit</a> <a href="#">Checkin</a> <a href="#">Cancel Checkout</a> <a href="#">Add To Clipboard</a> <a href="#">More...</a>	
	<a href="#">412a.pdf</a> Format: pdf Modified: 8/25/00 2:04 PM Edition: OK to Publish: False	<a href="#">Checkout</a> <a href="#">Edit</a> <a href="#">Checkin</a> <a href="#">Cancel Checkout</a> <a href="#">Add To Clipboard</a> <a href="#">More...</a>	

The object grid output when the new preference in the component definition is shown below. Only objects with webviewable set to true are shown:

**Figure 21-2. Component with preference**

	<a href="#">352intunixfcs.pdf</a> Format: pdf Modified: 1/9/02 1:57 PM Edition: Not Applicable OK to Publish: True	<a href="#">Checkout</a> <a href="#">Edit</a> <a href="#">Checkin</a> <a href="#">Cancel Checkout</a> <a href="#">Add To Clipboard</a> <a href="#">More...</a>	
	<a href="#">433Tutorial.zip</a> Format: zip Modified: 1/14/04 4:59 PM Edition: 4i OK to Publish: True	<a href="#">Checkout</a> <a href="#">Edit</a> <a href="#">Checkin</a> <a href="#">Cancel Checkout</a> <a href="#">Add To Clipboard</a> <a href="#">More...</a>	

To use the <preferences> element in the component definition, replace your custom element (<showwebviewableonly> in the example above) with one similar to the following:

```

<preferences>
 <preference id="showwebviewableonly">

```

```
<label><nlsid>MSG_PUBLISH</nlsid></label>
<description><nlsid>MSG_PUBLISH</nlsid></description>
<type>boolean</type>
<value>>true</value>
</preference>
</preferences>
```

In your component class, read the preferences element as follows. Note the lookup path notation for elements in the component configuration (highlighted):

```
private void readPrefs()
{
 Context cfgContext = new Context(Context.getApplicationContext());
 IConfigLookup lookup = ConfigService.getConfigLookup();
 Boolean bWebViewableOnly = lookup.lookupBoolean("
 component[id=webdocs].preferences.preference[id=
 showwebviewableonly].value", cfgContext);
 if (bWebViewableOnly != null)
 {
 m_WebViewableOnly = bWebViewableOnly.booleanValue();
 }
}
```

## Storing and retrieving component preferences

Use the preference lookup hook to store and retrieve preferences. The preference lookup hook routes all configuration service lookups through the preference service to check for a preference.

This lookup hook intercept calls to `lookupBoolean()`, and it adds the component path to the lookup. For example, the Login class reads its preferences with a call to `lookupBoolean()`:

```
Boolean bShowOptions = lookupBoolean(CONFIG_SHOWOPTIONS);
m_bShowOptions = bShowOptions.booleanValue();
```

The preference service appends the component path to the preference name: "component[login].showOptions". This full path name is passed to the preference store for lookup.

The component class then uses the preference in its business logic. In the same example, the Login class gets the preference value and passes it to a method that uses it:

```
showHideOptions(m_bShowOptions);
```

### Example 21-1. Overriding preferences

Default values for preferences can be defined in configuration XML files and can be overridden on a per-user basis. For example, the login component looks up the advanced options setting in the login dialog, which was automatically stored as a preference on a previous login:

```
Boolean bShowOptions = lookupBoolean("showOptions");
```

When you store a preference that overrides a component definition value, your component must write the preferences to the preference store with the same name that is used in the configuration path .

The following example overrides the advanced options selection in the login component. In a login class that extends Login, write the preference as follows:

```
IPreferenceStore store = PreferenceService.getPreferenceStore();
store.writeBoolean("component[login].showOptions", m_bAdvancedOptions);
```

As a shortcut to writing the full path, you can use the helper method `buildConfigPath` in the Component class. For example:

```
public MyLogin extends Component
{
 ...
 void setShowOptions(Boolean bShowOptions)
 {
 IPreferenceStore store = PreferenceService.getPreferenceStore();
 store.writeBoolean(buildConfigPath("showOptions"));
 }
}
```

## Storing and retrieving user preferences

User preferences are stored as cookies by default. The WDK preferences component is a container that displays several tabs, each generated by a specialized preferences component. The specialized preferences component displays several related preferences that affect one or more components. For example, in the general preferences component UI the user can select a preferred theme. The `AppGeneralPreferences` class gets the preference store and writes the preference.

`IPreferenceStore` provides an interface to the storage mechanism with methods for reading and writing preference values. The actual storage mechanism can consist of a file system, database, or cookies. (By default, WDK 5 stores preferences as cookies.) Only one instance of the preference store is used for each HTTP session.

The `PreferenceService` class uses `CookieManager` to store and retrieve preferences. `CookieManager` provides access to all cookies for the current session by calling `CookieManager.getCookieJar()`. Then you can use methods on the `CookieJar` class to add or remove cookies. Add a persistent cookie with `setCookie()` or a session cookie with `setSessionCookie()`. `CookieJar` concatenates all cookies into a single cookie and compresses it. The preferred method of storing preference cookies is using the `IPreferenceStore` methods described below.

Get the preference store by importing `IPreferenceStore` and `PreferenceService` into your class and then calling `getPreferenceStore()`:

```
import com.documentum.web.formext.config.IPreferenceStore;
import com.documentum.web.formext.config.PreferenceService;
...
IPreferenceStore preferenceStore = PreferenceService.getPreferenceStore();
```

When you create cookies, remember that HTTP cookie names and values cannot contain newline characters. The following `IPreferenceStore` methods read and write preferences:

- `readString(String strname)`: Reads a preference as a string. Returns the preference value as a `String`.
- `readBoolean(String strname)`: Reads a preference as a string. Returns the preference value as a `boolean`.
- `readInteger(String strname)`: Reads a preference as a string. Returns the preference value as an `integer`.
- `writeString(String strname, String strValue)`: Writes a preference name and string value.
- `writeBoolean(String strname, Boolean bValue)`: Writes a preference name and boolean value.
- `writeInteger(String strname, Integer nValue)`: Writes a preference name and integer value.

**Tip:** Preferences are stored as cookies. Since cookies are passed back and forth with every request and response, there is a small increase in network traffic. If you need to minimize network traffic or service users on low-bandwidth connections, you may wish to not use preferences. Additionally, cookies are stored on the browser's local machine, so preferences for roaming users is not supported.

To improve performance, cookie preferences are stored in memory the first time they are read or written in a session.

#### **Example 21-2. Storing user preferences**

The following example stores the user preference for displaying attributes:

```
IPreferenceStore store = PreferenceService.getPreferenceStore();
store.writeBoolean("component[id="attributes"].showAllAttributes", true);
```

The second example stores the user name as a `String`:

```
IPreferenceStore store = PreferenceService.getPreferenceStore();
store.writeString("Username", strUserName);
```

**Note:** When you create preferences, remember that HTTP cookie names and values cannot contain newline characters.

#### **Example 21-3. Retrieving user preferences**

By default, the `readXXX` methods return a reference to the cookie-based preference store.

In the following example, `CancelCheckout` reads the preference store in order to display a warning:

```
public static boolean showChangeLossWarning()
```



```
{
 IPreferenceStore preferences = PreferenceService.getPreferenceStore();

 if (preferences.readBoolean(INHIBIT_CHANGE_LOSS_WARNING) == null)
 {
 // preference to not warn has not been stored
 return true;
 }
 else
 {
 // preference to not warn has been stored
 return false;
 }
}
```

The following example gets the user name preference:

```
//Get a username preference value
IPreferenceStore store = PreferenceService.getPreferenceStore();
String strUsername = store.readString ("Username");
```

## Tracing preferences

To turn on preferences tracing, set the following flags in `com.documentum.debug.TraceProp.properties` (located in `/WEB-INF/classes`):

```
com.documentum.web.formext.Trace.CONFIGSERVICE=true
com.documentum.web.formext.Trace.PREFERENCES=true
```

You can also turn on preferences tracing by navigating to `http://application_name/wdk/tracing.jsp`.



## Other Customizations

The WDK 5 framework consists of services that are used by more than one component in the application. For example, the preferences service is used by the several components and can potentially be used by any component. The configuration, action, role, and content transfer services are discussed in their own chapters. (Refer to [Chapter 14, Using the Configuration Service](#), [Chapter 15, Customizing actions](#), [Chapter 16, Customizing Roles](#), and [Chapter 17, Customizing Content Transfer](#), respectively). The framework also provides the following services:

- [Asynchronous action and component execution, page 571](#)
- [Branding service, page 578](#)
- [Image service APIs, page 579](#)
- [Locale service, page 580](#)
- [Accessibility service, page 584](#)
- [Help service, page 590](#)
- [Utilities, page 595](#)

## Asynchronous action and component execution

The asynchronous framework in WDK allows component and action jobs to run asynchronously, returning control to the client immediately. An example of a job is either the component event handler code or action code to be executed.

You can turn on or off asynchronous processing support for the application in the application's app.xml file. In this file you can also specify a global job event handler for pre- and post-processing, set the maximum number of asynchronous jobs per user, and turn on or off user notification of job finish. The event handlers and job notification settings are overridden by settings in your asynchronous component or action definition. Global asynchronous settings in app.xml are described in [Table 2–33, page 83](#).

The asynchronous framework has the following features:

- Component and action jobs can be run synchronously or asynchronously
- Asynchronous support can be turned on or off globally
- Details of running asynchronous components and actions can be viewed
- Asynchronous execution can be aborted from the UI
- The user inbox receives a notice upon asynchronous completion (finished, failed, aborted)
- Handlers are called to perform pre- and post-processing of asynchronous jobs
  - The pre-execution callback handler can call UI components and indicate whether to proceed with execution
  - A global pre- or post-execution handler can be specified in app.xml. This handler can be overridden in the action or component definition.

The following sections describe asynchronous support:

- [Asynchronous action job execution, page 572](#)
- [Asynchronous component job execution, page 574](#)
- [Job execution framework, page 575](#)
- [UI in asynchronous processing, page 577](#)
- [Asynchronous process, page 577](#)

## Asynchronous action job execution

Custom actions based on WDK 5 will work in the 5.2.5 asynchronous framework. By default, all actions execute synchronously unless configured as asynchronous.

When an action is invoked from the UI, the action service is called to invoke the action implementation. The action implementation calls the job execution service to execute the job. If the job is asynchronous, the job execution service calls the asynchronous job manager to execute the job asynchronously.

Action completed listeners are called when an action is started asynchronously. The thread for the asynchronous action calls the post-processing handler after the asynchronous action has completed.

To enable asynchronous execution of an action, you must perform the following steps:

1. Add to the action definition an `<asynchronous>` element as a child of the `<action>` element with a value of true.

If the `<asynchronous>` element is not present, the value is assumed to be false, and the action will execute synchronously.

2. (Optional) Set the `<asynchronous>` element attribute `sendnoticeonfinish` to true to notify the user inbox when the action is finished.

3. (Optional) Add a pre- and/or a post-execution handler to the action definition by adding a `<job-event-handler>` element whose value is the fully qualified class name of the event handler.
4. Add action implementation code to your action class that calls the job execution service.
5. Add the internal job class to your action implementation

#### Example 22-1. Enabling asynchronous execution of an action

The following example enables asynchronous execution of a delete action.

1. Add an `<asynchronous>` element with a value of true to the action definition:

```
<action id="delete">
 ...
 <asynchronous sendnoticeonfinish="false">true</asynchronous>
 ...
</action>
```

2. (Optional) Add a pre- and/or a post-execution handler to the action definition:

```
<job-eventhandler>com.documentum.custom.DeleteHandler</job-eventhandler>
```

3. Add action implementation code to your action class. Note that the arguments are passed to the internal Job implementation (highlighted):

```
package com.documentum.test;

import com.documentum.web.formext.action.IActionExecution;
import com.documentum.web.formext.config.IConfigElement;
import com.documentum.web.formext.config.Context;
import com.documentum.web.formext.component.Component;
import com.documentum.web.common.ArgumentList;
import com.documentum.web.common.job.JobExecutionService;
import com.documentum.job.Job;
import com.documentum.fc.client.IDfSessionManager;

public class TestAction implements IActionExecution
{
 public boolean execute(String strAction, IConfigElement config,
 ArgumentList args, Context context, Component component,
 Map completionArgs)
 {
 return JobExecutionService.getInstance().executeActionJob(
 new TestActionJob(args), args, context,
 component, strAction, null);
 }

 public String[] getRequiredParams()
 {
 return new String[0];
 }
}
```

4. Add the internal job class implementation to your action implementation:

```
private static class TestActionJob extends Job
{
 // Get arguments
 public TestActionJob(ArgumentList actionArgs)
 {
 m_actionArgs = actionArgs;
 }

 public boolean execut(IDfSession Manager sessMgr,
 String docbaseName)
 {
 System.out.println("Thread name:" + Thread.currentThread(
).getName());
 // use the m_actionArgs here
 return true;
 }

 public String getName()
 {
 return "test action job";
 }

 private ArgumentList m_actionArgs;
}
```

## Asynchronous component job execution

Custom components based on WDK 5 will work in the 5.2.5 asynchronous framework. By default, all components execute synchronously unless configured as asynchronous.

When a component event handler is invoked from the component UI (JSP page), the event handler calls the job execution service to execute the job. If the job is asynchronous, the service calls the asynchronous job manager to execute the task asynchronously.

To enable asynchronous execution of a component job, you must perform the following steps:

1. Add an <asynchronous> element with a value of true to the component definition. The parent element is <component>. If this element is not present, the value is assumed to be false, and the component will execute synchronously. Set the attribute sendnoticeonfinish to true to notify the user inbox when the component job is finished.
2. (Optional) Add a pre- and/or a post-execution handler to the component definition by adding a <job-event-handler> element whose value is the fully qualified class name of the event handler.
3. Add component implementation code to your component class that calls the job execution service.

4. Add to the component class an inner class that extends `com.documentum.job.Job`.

**Example 22-2. Enabling asynchronous execution of a component**

The following example enables asynchronous execution of checkin component.

1. Add an `<asynchronous>` element with a value of true to the component definition:

```
<asynchronous sendnoticeonfinish="false">true</asynchronous>
```

2. (Optional) Add a pre- and/or a post-execution handler to the component definition:

```
<job-eventhandler>com.documentum.custom.DeleteHandler</job-eventhandler>
```

3. Add component implementation code to your component class:

```
onCommitChanges()
{
 CheckinJob job = new CheckinJob();
 JobExecutionService service = JobExecutionService.getInstance();
 service.execute(job, args, getContext(), this);
}
```

4. Add to the component class an inner class that extends `com.documentum.job.Job`:

```
Private class CheckinJob extends Job
{
 Public void execute()
 {
 //code to perform checkin operation
 //report progress using Job.setStatusReport()
 }
}
```

## Job execution framework

The package `com.documentum.jobs` is responsible for asynchronously executing action and component jobs. The caller can either pull or be pushed with the status and the progress of the async jobs. The caller can also pass a job lifecycle event handler. Appropriate methods in the event handler will get called during the job execution lifecycle. The job interacts with the caller by suspending its execution and asking the caller to provide input data to continue the execution. The caller can either ask the job to continue after providing the data or simply ask the job to abort the execution.

The job implementation can break the whole task into a set of subtasks called steps. Your implementation can specify the number of steps and the names of each step. A progress bar in the UI then displays progress for each step. The job implementation must keep track of which step it is in during the execution.

**Example 22-3. Job with Steps**

In the following example, an asynchronous action class adds steps:

```
import com.documentum.job.Job;
```

```
import com.documentum.fc.client.IDfSessionManager;

public class JobWithSteps extends Job
{
 public JobWithSteps()
 {
 // When adding steps, the base Job implementation keeps track of the
 // number of steps. The steps could also be read from a properties file
 addStep("Initializing"); // step 1
 addStep("Reading data"); // step 2
 addStep(""); // step3 (the step name is unknown during init)
 addStep("Transforming data"); // step 4
 }

 /**
 * Executes the job.
 * @return True if the execution is successful; false otherwise
 */
 public boolean execute(IDfSessionManager sessionManager, String docbaseName)
 {
 // by default the job is in step 1
 // perform step 1 logic here. The status report is filled up with step number 1,
 // the step name, and total # of steps

 nextStep(); // now the job is in step 2

 // perform logic for step 2. The status report is filled up with step number 2,
 // the step name, and total # of steps

 nextStep(); // now the job is in step 3

 // In the constructor, the step name of step 3 is set as empty string.
 // Update the step name here
 setStepName(getCurrentStep() - 1, "Parsing data");

 nextStep(); // now the job is in step 4
 // perform step 4 logic. The status report is filled up with step number 4,
 // the step name, and total # of steps

 return true;
 }

 /**
 * Returns the name of the job
 * @return Display name of the job
 */
 public String getName()
 {
 return "Test Job";
 }
}
```



## UI in asynchronous processing

The UI for invoking an asynchronous action or component is no different from the UI to invoke synchronous jobs. The following UI components or controls are used to inform the user of asynchronous job processing:

- Display task status icon in the statusbar component

The Webtop statusbar component adds an animated job status icon to display progress and a button to launch the jobstatus component. The icon will be displayed dynamically when asynchronous jobs are running.

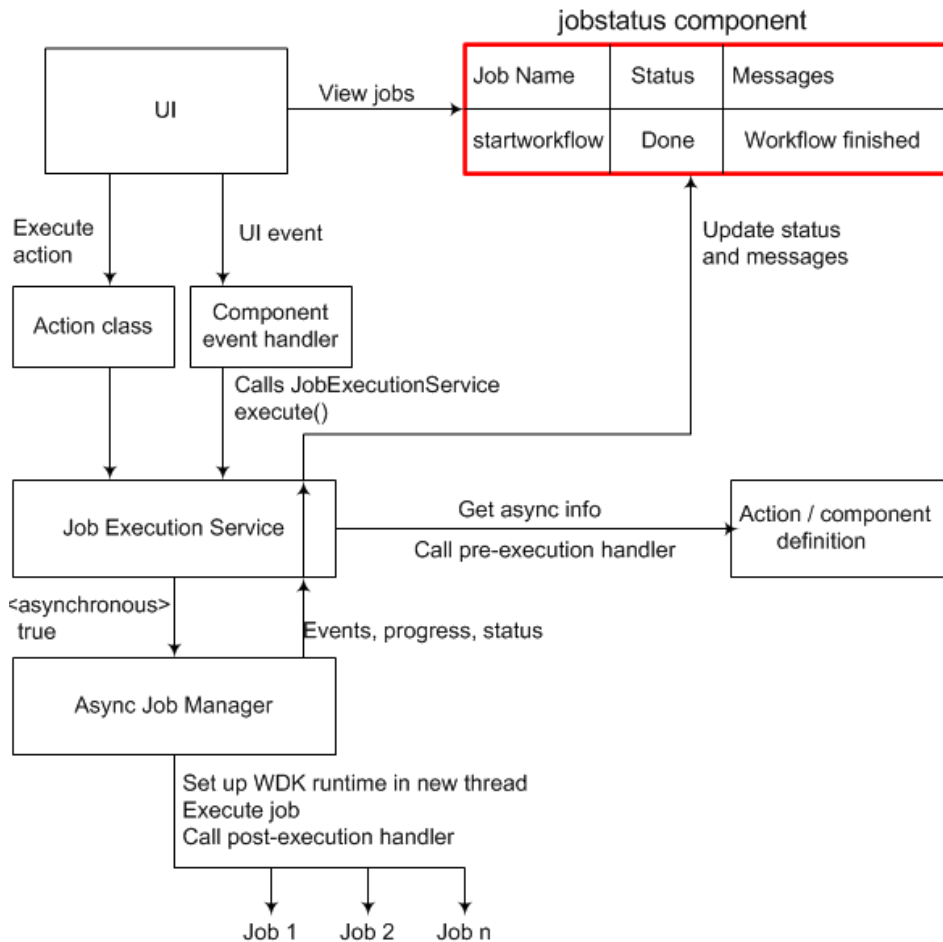
- Display the status of an asynchronous job

The jobstatus component displays the job details such as component or action name, current status, and messages generated by the component or action. This component uses the StatusListener class to provide status details.

## Asynchronous process

The process in which asynchronous actions and component jobs execute is diagrammed below:

Figure 22-1. Job execution interaction diagram



## Branding service

The branding service allows you to customize the look of your application user interface. The branding service manages the UI look by themes, which incorporate images and icons, and cascading style sheets (CSS).

Users select a theme for display in the preferences component. The set of themes available are configured in `/wdk/app.xml`. The logic for theme lookup of registered theme is in the class `ConfigThemeResolver`. To override the default theme lookup from `app.xml`, implement your own theme resolver based on user role or some other criterion. Implement `IThemeResolver` and register your resolver in the properties file `/WEB-INF/classes/com/documentum/web/common/BrandingServiceProp.properties`.

Additionally, you can override the theme that is displayed dynamically, based on user roles or some other criterion.

The [Image service APIs, page 579](#) provides support for loading and caching style sheets, files, and images.

## Image service APIs

Using the image service, you can manage the images used as background for buttons, labels, and other controls. You can resolve icon state programmatically or based on Documentum attributes. Use the image service to determine whether the referenced image exists on the app server and to cache the image dimensions for faster loading.

The image service enhances the performance of pages containing images by fetching images from the local file system of the app server and caching the image and the image dimensions. The image service uses the file checker service to determine whether the file exists on the J2EE server file system.

Graphical tab bars use the image service to set the height of the tab bar, based on the heights of images in the bar. Tree control images use the image service to determine the dimensions of tree icons. Tree nodeheight and nodewidth values can be overridden in the JSP tag.

The ImageService class has the following APIs:

**isExist(String strUrl, PageContext pageContext)** — Tests whether a given URL exists on the file system. This method is a wrapper for FileChecker.isExist. Use this to construct an image in your tag class.

### Example 22-4. Getting an image in a tag class

In the following example from ActionMenuItemTag, the image service provides the required image:

```
private void renderImageIcon(StringBuffer buf, ActionMenuItem menuItem)
{
 String strImageIcon = menuItem.getImageIcon();
 // check that the image icon exists
 if (ImageService.isExist(strImageIcon, pageContext) == false)
 {
 strImageIcon = null;
 }
 else
 {
 // prepend virtual root
 strImageIcon = Form.makeUrl(pageContext.getRequest(), strImageIcon);
 }
 // render
 if (strImageIcon != null)
 {
```

```
 buf.append('\n')
 .append(strImageIcon)
 .append('\n');
 }
 else
 //handle missing image
 }
```

**getDimensions(String strUrl, PageContext pageContext)** — Takes a URL to a GIF image and returns the height and width of the image as a `java.awt.Dimension` object. If the dimensions have not been previously cached, the image service calls the file checker service. Use the dimensions to render an image.

#### Example 22-5. Getting the dimensions of an image

In the following example from the `BookmarkLinkTag` class, the image service provides the image width and height for output to HTML:

```
Dimension extents = ImageService.getDimensions(
 strBookmarkIconUrl, getForm().getPageContext());

buf.append("<img src='")
.append(SafeHTMLString.escape(strBookmarkIconUrl))
.append("' width=")
.append((int)extents.getWidth())
.append(" height=")
.append((int)extents.getHeight())
.append(" alt='")
.append(SafeHTMLString.escape(getTooltipString(link)))
.append("' border=0/>");
```

## Locale service

WDK supports localization (translation) of the UI through the locale service and National Language Service (NLS) lookup. Locale support is specified in the application configuration file. When the user selects a locale, the appropriately-named set of localized strings will be used. The localized strings are contained in NLS properties files. For more information on creating, naming, and adding localized files to your application, refer to [Configuring and localizing strings, page 137](#).

Applet strings are externalized, and string values are passed to the applets via HTML parameters that are generated by the content transfer applet tags.

Images have an NLS entry in a resource bundle. This string is displayed as the HTML image alt tag text to support accessibility requirements. Refer to [Image accessibility strings, page 587](#) for information on configuring image alt tags.

The following topics describe customization involving the locale service or NLS bundles:

- [Retrieving localized strings, page 581](#)

- [Dynamic messages in NLS strings, page 582](#)
- [Adding locale support to custom components, page 583](#)
- [LocaleService APIs, page 583](#)
- [Locale codes, page 584](#)

## Retrieving localized strings

The component definition specifies the name of the NLS bundle or NLS class to be used for lookup. For example:

```
<nlsbundle>com.documentum.example.ExampleNlsProp</nlsbundle>
```

The locale service looks up the resource file in `/custom/strings/documentum/example/ExampleNlsProp.properties`. The string is dereferenced from the properties file:

```
MSG_EXAMPLE=This is an example
```

If you reference a string with its NLS ID in a JSP page or Java class, the configuration service will look up the `nlsid` element in a properties file for the user's locale. The following example retrieves a string from `/WEB-INF/classes/com/example/MyComponentNlsProp.properties`:

```
<dmf:webform/>
<dmf:label name="label1" nlsid="MSG_EXAMPLE");
```

For general uses you can retrieve one your component strings in the following way:

```
<%
 CustomComponentName form = (
 CustomComponentName)pageContext.getAttribute(
 Form.FORM, PageContext.REQUEST_SCOPE);
%>
<span title='<%=form.getString(
 "MSG_COMPONENT_STRING")%>'>
```

To retrieve a localized string in your component Java class, call the `getString()` method, passing in the NLS ID. This method will look for the resource file named in the component definition and deference the string that is represented by the NLS ID:

```
import java.util.ResourceBundle;
...
String strExample = getString("MSG_EXAMPLE");
demoExample.setLabel(strExample);
```

You can use the `NlsResourceClass` and `NlsResourceBundle` method `stringExists()` to determine whether a string exists.

## Dynamic messages in NLS strings

You can write messages from your action or component class using `MessageService` that take a runtime message and add it to an introductory NLS string. In your component or action class, import `MessageService` and add the message, similar to the following:

### Example 22-6. Dynamic Error Messages

In the following example from `SubmitForCategorization`, two different messages are dispatched depending on the runtime context:

```
import com.documentum.webcomponent.library.messages.MessageService;
...
IDfSysObject oObject = getObject();
if (m_fIsAutoEnabled)
{
 oObject.queue(
 DEF_USER_AUTO_PROC, QUEUE_EVENT_AUTO_PROCESS, QUEUE_PRIORITY,
 QUEUE_IS_MAIL, new DfTime(), getString("MSG_QUEUE_AUTO_PROC"));
 MessageService.addMessage(
 this, "MSG_ACKNOWLEDGE_AUTO_SUBMISSION", new String[] {
 oObject.getObjectname()});
}
else
{
 oObject.queue(
 DEF_USER_MANUAL_PROC, QUEUE_EVENT_MANUAL_PROCESS, QUEUE_PRIORITY,
 QUEUE_IS_MAIL, new DfTime(), getString("MSG_QUEUE_MANUAL_PROC"));
 MessageService.addMessage(
 this, "MSG_ACKNOWLEDGE_MANUAL_SUBMISSION", new String[] {
 oObject.getObjectname()});
}
```

The third parameter to `addMessage()` is an array of parameters for runtime substitution in the NLS string. The array does not have to be object name as in the example above.

`MessageService` uses `java.text.MessageFormat` to perform the substitution. Refer to the J2SE javadocs for `MessageFormat` for more information on concatenating dynamic messages.

The properties file has an entry similar to the following:

```
MSG_ACKNOWLEDGE_MANUAL_SUBMISSION=
 The document "{0}" was submitted to the queue for manual
 categorization.
```

**Tip:** WDK finds the message string by the component definition. The component in this example, `submitforcategorization`, names the NLS bundle that contains its strings, `com.documentum.webcomponent.library.submitforcategorization.SubmitForCategorizationNlsProp`.

## Adding locale support to custom components

Your components must specify in their configuration file an NLS bundle (<nlsbundle>) that will be used to look up strings based on user locale. For example:

```
<config>
<scope type='dm_sysobject'>
<component id='mycomponent'>
 <nlsbundle>com.example.mycomponent.MyComponentNlsProp</nlsbundle>
 ...

```

Create the properties file corresponding to your bundle class. For example:

```
MSG_TITLE=Demo
MSG_EXAMPLE=This is your example

```

Add the properties files to the /strings directory under your custom application directory. Properties files in an application that extends another application override the string definitions in the base application.

## LocaleService APIs

The locale service is implemented in the LocaleService class. This class has the following methods:

**getLocale()** — Retrieves Locale from the session. If the session variable doesn't exist, it will be created.

### Example 22-7. Getting the user's locale

The locale service can retrieve the user's locale. In the following example from DynamicPerformerResultSet, getLocale() retrieves a localized string:

```
// initialize values coming from the bundle
Locale locale = LocaleService.getLocale();
m_strNullPerformer = getNlsResourceClass().getString(
 "MSG_NO_PERFORMER_ASSIGNED", locale);
m_delim = getNlsResourceClass().getString(
 "MSG_MULTIPLE_PERFORMER_NAME_DELIMITER", locale);

```

**setLocale()** — Sets a new locale in the session.

### Example 22-8. Setting the locale for an object

In the following example from the Web Publisher class WcmContent, setLocale() sets the locale for a new translation:

```
theDoc.setObjectName(name);

```

```
// need to set this before we promote the object to WIP state
theDoc.setLocale(newLocale);
```

**getDefaultLocale()** — Returns the default locale based on the locale of the application server OS.

**getSupportedLocales()** — Gets the locales that are named in app.xml.

**createLocale()** — Creates a Locale object from a string representation of a locale.

## Locale codes

The WDK locale service uses Java locale names and country codes to set the user's locale.

Supported locales are configured in the application app.xml file. In your custom layer app.xml file you can override the <supported\_locales> element and list the locales that are supported by your application.

Java locales are constructed from a concatenation of the two-letter ISO language code and the two-letter ISO country code in the form xx\_YY, where xx is the two-character lower-case language code and YY is the two-character uppercase code. The language code alone (YY) is an acceptable locale code string.

For the full list of ISO language codes, refer to [Code for the Representation of the Names of Languages. From ISO 639](#). For the full list of ISO country codes, refer to [English country names and code elements](#).

## Accessibility service

WDK components are compliant with the standards of section 508 of the U.S. Disabilities Act. The accessibility service provides support for displaying a UI that is accessible to vision-impaired users. The service turns on accessibility based on the user's selection of accessibility in the login page. The accessibility preference is stored in a cookie along with other user preferences.

The accessibility service turns on support for keyboard navigation, tooltip presentation, and special navigation pages that are rendered in place of actionmultiselectcheckbox controls. For information on configuring accessibility features, refer to [Configuring accessibility, page 310](#).

The following topics describe WDK accessibility features and how to use them to create an accessible Web application:



- [Accessibility mode, page 585](#)
- [Accessible control labels, page 586](#)
- [Event handlers, page 586](#)
- [Image accessibility strings, page 587](#)
- [Accessible tables, page 588](#)
- [Applet descriptions, page 589](#)
- [Frame titles, page 589](#)
- [Writing alt tags or label descriptions, page 590](#)

## Accessibility mode

In accessibility mode, several controls and components have different behavior:

- HTTP content transfer is used for import when the user has selected the accessibility mode.
- The `actionmultiselectcheckbox` control is rendered as a link to a list of actions for the object.
- The `actionmultiselectcheckall` control is rendered as a link to a page of global actions, that is, actions that do not require an object.
- Menu controls (`menugroup`, `menuitem`, `actionmenuitem`, `menuseparator`) generate a single link that invokes the action that is normally associated with the menu item.
- The Webtop menubar component renders a page of action links for an item. It does not render a menu bar.
- The help service launches a PDF file, which is accessible, instead of online help.
- Button controls have an `accessible` parameter. When set to true, the button is rendered as a link that is accessible by the tab key. This parameter can be set independently of the user's accessibility preference and is used to make the buttons on the login page accessible before the accessibility option is selected.
- The login component page has tab-accessible buttons and a checkbox that enables accessibility for the user.
- The Webtop browsertree and titlebar components have an additional link to the user's work area at the beginning of all links on the page. This allows the user to tab to the link and navigate to another frame directly. The shortcut to the workarea is provided through a JavaScript function, `setFocusOnFrame(Frame framename)`. This JavaScript function is contained in the file `/wdk/include/locate.js`.
- Accessibility defaults for alt text, keyboard navigation, and shortcuts can be set in the application configuration file `app.xml`. Refer to [Table 2-9, page 65](#) for details.
- Alt text strings can be configured for images. Refer to [Image accessibility strings, page 587](#) for information on how to use these strings in a WDK-based application.

**Note:** Multiple selection is not supported in accessibility mode. Instead, a tab-accessible link next to each object launches an action page of actions that are available for the object.

## Accessible control labels

Every control must be labelled, either by using one of the tooltip attributes, if supported, or by adding a label tag. A non-compliant control has neither kind of label:

```
<dmf:text name="containedwords" focus="true" size="40"
 defaulttonenter="true"/>
```

The same control can have a tooltip attribute. Many controls support this attribute, and some additionally support tooltipnlsid and tooltipnlsdatafield. (Refer to the tag library descriptor for supported attributes on a particular control.):

```
<dmf:text name="containedwords" focus="true" size="40"
 defaulttonenter="true" tooltipnlsid="MSG_CONTAINING"/>
```

Alternatively, a control can have an associated label tag, which supports tooltips:

```
<LABEL for="Email"> Email </LABEL>
<INPUT type="text" name="emailinput" size="8" class="NavBold" id="Email">
</INPUT>
```

## Event handlers

Event handlers must not be device dependent, that is, dependent on mouse input. A non-compliant handler is similar to the following:

```
<a href="http://www.yahoo.com" onmouseover="window.status=
 'Go to the Yahoo homepage':return true">Yahoo Home Page
```

The same event can be handled in a device-independent manner by adding an onFocus event:

```
<a href="http://www.yahoo.com" onMouseOver="window.status=
 'Go to the Yahoo homepage':return true" onFocus="window.status=
 'Go to the Yahoo homepage':return true">Yahoo Home Page
```

The following table lists device-independent alternatives to mouse events:

**Table 22-1. Device-independent events**

Mouse event	Device-independent event
onClick	onKeyPress
onMouseDown	onKeyDown

Mouse event	Device-independent event
onMouseUp	onKeyUp
onMouseOver	onFocus
onMouseOut	onBlur
onDbIcIck	onKeyDown

## Image accessibility strings

Images can be rendered with HTML alt attribute text values to comply with accessibility standards. A non-compliant example of an image tag is:

```
IMG src="library_map.gif"
```

The image can be made accessible by adding a properties file with alt text for the image. The WDK image accessibility resource files are located in the following directory of each application layer: `/strings/com/documentum/web/layer_name/accessibility/icons` and `/strings/com/documentum/web/layer_name/accessibility/images` where `layer_name` is `wdk`, `webcomponent`, `webtop`, `wp`, `custom`, or a client application layer name.

### To define the lookup for an image or icon file or directory

You can create a lookup file for a single image or all of the images in a directory, but not the subdirectories. In the following example, you are creating the alt text for a several format icons in a theme directory (the first icon is named `f_123w_16.gif`).

1. Create a properties file named `alt.properties` in the same directory as the image.
2. Specify the properties file for the image (icon) or images (icons). For example:

```
nlsbundle=com.documentum.web.accessibility.icons.FormatAltNlsProp
```
3. Create the properties file that you referenced in the `alt.properties` file. In the above example, you would create a file `/WEB-INF/classes/com/documentum/web/accessibility/icons/FormatAltNlsProp.properties`.

Add a key to the property file for each image or icon. For example:

```
f_123w_16.gif = Lotus 1-2-3 r5
```

To override the alt text displayed for images and icons within a component, redefine the NLS lookup in the component NLS file.

## Accessible tables

To be accessible, tables must have titles or be described just before the table is presented. Additionally, tables must have header cells that describe the contents of each column and row scope cells at the beginning of each row. The following table is not compliant because reader software cannot distinguish which column a table cell belongs to:

```
<TABLE>
 <TR>
 <TD>Jane</TD>
 <TD>20</TD>
 <TD>Female</TD>
 </TR>
 <TR>
 <TD>John</TD>
 <TD>23</TD>
 <TD>Male</TD>
 </TR>
</TABLE>
```

The above example can be made accessible with the addition of table header cells with ID, matched to a headers attribute on each table cell. This method is useful for small tables:

```
<TABLE>
 <TR>
 <TH id="names">Name</TH>
 <TH id="ages">Age</TH>
 <TH id="sex">Sex</TH>
 </TR>
 <TR>
 <TD headers="names">Jane</TD>
 <TD headers="ages">20</TD>
 <TD headers="sex">Female</TD>
 </TR>
 <TR>
 <TD headers="names">John</TD>
 <TD headers="ages">23</TD>
 <TD headers="sex">Male</TD>
 </TR>
</TABLE>
```

Alternatively, you can use a column scope attribute on each table header cell and a row scope attribute on the header cell of each table row. This method is supported by most but not all accessibility software. The first cell must contain identifying information for the row:

```
<TABLE>
 <TR>
 <TH scope="col">Name</TH>
 <TH scope="col">Age</TH>
 <TH scope="col">Sex</TH>
 </TR>
 <TR>
```

```

<TH scope="row">Jane</TD>
 <TD>20</TD>
 <TD>Female</TD>
</TR>
<TR>
 <TH scope="row">John</TD>
 <TD>23</TD>
 <TD>Male</TD>
</TR>
</TABLE>

```

## Applet descriptions

APPLET elements should contain a text equivalent, usually in the form of an alt attribute. Some assistive technologies and browsers cannot understand their content. In addition, the interface for the applet itself must be accessible.

A non-compliant applet tag has no description, similar to the following:

```
<APPLET code="urname.class" width="10" height="10"></applet>
```

The above example can be made accessible with the addition of the alt attribute and a text string description as follows:

```

<APPLET code="urname.class" width="250" height="22"
 alt="This applet allows a user to view a clock">
 This applet allows a user to view a clock
</APPLET>.

```

## Frame titles

HTML <frame> tags should have titles. Use the WDK frame tags to generate a frame with a title and ID. The ID is necessary to maintain proper browser history and state.

The following example generates a non-accessible frame:

```

<dmf:frame nlsid="MSG_MESSAGEBAR" frameborder="false"
 name="messagebar" src="/component/messagebar" scrolling="no"
 noresize="true"/>

```

The above example is compliant with the addition of a frame title attribute:

```

<dmf:frame nlsid="MSG_MESSAGEBAR" title="Message Bar"
 frameborder="false" name="messagebar" src="/component/messagebar"
 scrolling="no" noresize="true"/>

```

## Writing alt tags or label descriptions

The basic formula for a Documentum ALT tag or label is Noun+Ver, for example: Element name + What the element does. Visually impaired users who are familiar with the UI can hear the name from the screen reader and proceed. A full but not lengthy description following the name will help less familiar users and will not impede more experienced users.

Use a period between the element name and the description. This will cause the screen reader software to pause and allow the experienced user to move on unless they wish to hear the description.

The following example is the alt text for the inbox graphic:

**Documentum Inbox. Where you can check your workflow tasks and notifications.**

## Help service

For standalone Web applications, the help service enables context-sensitive help at the component level. For portal applications, context-sensitive help is configured through the portlet configuration file `portlet.xml` in `/WEB-INF`. When the user launches the help button or help link, the help for the current component is displayed in a help host page (in a separate browser window).

The help service is described in the following topics:

- [Adding help to a standalone Web application, page 590](#)
- [Localizing help files, page 595](#)

## Adding help to a standalone Web application

You can use WDK component help in your application and add new help files for your custom components. A set of help files is included in the WDK client applications in the `/help` directory at the root of application.

To add help files for your custom application, first install WDK and select the option to customize an existing application. Select an installation of Webtop or another WDK-based Web application. The application will be copied to a development directory in which you can customize the application and the help files.

The following topics provide details on the help component:

- [The help component, page 591](#)
- [Adding help for a custom component, page 591](#)

- [Invoking the help, page 592](#)
- [Scoping and filtering the help, page 593](#)
- [Launching help, page 594](#)

## The help component

Help files are mapped in the help-index component definition. The configuration file that contains this definition, `help_component.xml`, is located in the product application layer `/config` directory or a subdirectory of the `/config` directory. For example, if you are customizing the Webtop product, the help-index component definition is located in `/webtop/config`. If you are customizing the Web Publisher product, the definition is located in `/wp/config/app`.

The names of help files are specified in the `<help-entries>` element. Each file name is specified as the value of an `<entry>` element. The `id` attribute of the `<entry>` element matches the value of the `<helpcontextid>` element in a component definition.

For example, the entry for the `abortwpworkflow` component in Web Publisher is specified as follows.

```
<entry id="abortwpworkflow">wp_managing_workflows.htm</entry>
```

The entry `id` matches the `helpcontext` value in the `abortwpworkflow` definition:

```
<helpcontextid>abortwpworkflow</helpcontextid>
```

## Adding help for a custom component

When you write a new component and need to provide end-user help, you should extend the help-index component so that your application can be updated without overwriting your custom help files.

**Note:** The help file must be in a format that can be viewed in a browser, such as HTML or PDF.

If your custom component extends the functionality of a WDK component, and the WDK help is sufficient, the WDK help will be displayed by the help service, because your component inherits the `helpcontextid` value from the parent component. You do not need to extend the help component.

If your component provides new functionality that has separate documentation, you can copy the WDK help topic for the parent component or create your own HTML file, using a WDK help file as a template.

### To add help for a new component

1. Create a helpcontextid for your component in the component definition. For example:

```
<component id="my_component">
 ...
 <helpcontextid>my_component</helpcontextid>
</component>
```

2. Extend the help-index component.
  - a. Copy the help-index component definition from /config/app to /custom/config.
  - b. Change the component element to extend the original. For example (substitute the appropriate product directory, such as wp or dam):

```
<component id="help-index" extends=
 "help-index:webtop/config/app/help_component.xml">
```

3. Map the component to a help file in your custom help-index component definition, adding an entry for each help file. For example:

```
<entry id="my_component">my_component.htm</entry>
```

4. Add your localized help files for each component to the appropriate directories. For example, if you have localized my\_component.htm in French, Spanish, and English, add the copies to /help/en and /help/fr. Make sure you have added an entry for the locales in app.xml.

## Invoking the help

The help is invoked when a button, link, or other control in your application calls the JavaScript function `onClickHelp()` in the file `/wdk/include/help.js`. A new window is launched containing the context-appropriate help file.

If your component will be used for both portal and standalone environments, name your help button `DialogContainer.CONTROL_HELPBUTTON`. (You must import `com.documentum.web.formext.component.DialogContainer` into your JSP page.) Help buttons with this name will be suppressed in portal environments so that portlet help can be launched instead. For example:

```
<dmf:button name='<%=DialogContainer.CONTROL_HELPBUTTON %>' nlsid='MSG_HELP'
 onclick='onClickHelp' runatclient='true' height='16' cssclass="buttonLink"
 imagefolder='images/dialogbutton' tooltipnlsid="MSG_HELP_TIP"/>
```

To display help for any control that has an `onclick` attribute, set the attribute to call `onClickHelp()`. Some controls that can launch the help are: button, link, radio, checkbox, image, tab, and menuitem. For example:

```
<dmf:button name="help" cssclass="buttonLink" nlsid="MSG_HELP"
 onclick="onClickHelp" ...>
</dmf:button>
```



The help for the component that contains the control will be displayed.

The supporting JavaScript `onClickHelp()` function is available to every JSP page that contains the `<dmf:webform/>` tag. This tag generates in the HTML output a reference to the `help.js` JavaScript file .

## Scoping and filtering the help

You can set the help context based on the value of a scope qualifier. Alternatively, you can add a filter to the help that displays the help when a qualifier value is satisfied. The scope qualifier is applied in the component definition. The filter qualifier is applied in the `help-index` definition.

### Example 22-9. Scoping the help

When you scope a component based on a qualifier, you can provide a different `helpcontextid` value for that scope. In the following example, the administrator and consumer users see different help files:

```
<config>
<scope role="administrator">
<component id="subscriptions_list">
 ...
 <pages>
 <start>subscriptions_admin.jsp</start>
 </pages>
 <helpcontextid>subscriptions_admin</helpcontextid>
</component>
</scope>

<scope role="consumer">
<component id="subscriptions_list">
 ...
 <pages>
 <start>subscriptions_consumer.jsp</start>
 </pages>
 <helpcontextid>subscriptions_consumer</helpcontextid>
</component>
</scope>
</config>
```

Your `help-index` component definition has two entries:

```
<entry id="subscriptions_admin">subscriptions_admin.htm</entry>
<entry id="subscriptions_consumer">subscriptions_consumer.htm</entry>
```

### Example 22-10. Filtering the help files

You can filter a set of help files that are displayed only when a qualifier value is satisfied. For example, the Web Publisher application is built on Webtop and displays Webtop

help. The Web Publisher help-index component adds a set of help entries that are displayed only for a certain qualifier value. The Web Publisher components all have a `wpcontext` value of `wpview`. You can use any qualifier that is defined in your application to filter help files.

In the following example, you have different properties help depending on the repository, but the help refers to a single component. Your help-index component definition would look something like this:

```
<component id="help-index" extends=
 "help-index:webtop/config/app/help_component.xml">
 ...
 <help-entries>
 <entry id="default">default.htm</entry>
 <!-- Docbase A1 helps -->
 <filter docbase="A1">
 <entry id="properties">properties_a1.htm</entry>
 </filter>
 <!-- Docbase A2 helps -->
 <filter docbase="A2">
 <entry id="properties">properties_a2.htm</entry>
 </filter>
 ...
 </help-entries>
```

## Launching help

The base directory for help is registered in the properties file `com.documentum.help.helpProp.properties`. The default base directory for the help is `web_root/help`.

The help service uses a base URL, which is registered in `/WEB-INF/classes/com/documentum/web/common/HelpService.properties`, to locate the JSP page that hosts the help file. The base URL is defined in the properties file as follows:

```
HelpUrl=/help/help.jsp?context=
```

The value of the context is added by the help service based on the current component.

The help host page detects the client browser and launches the context-sensitive help file for the current component in a new window. The help service looks up the context value in the help-index component definition and launches the appropriate help page for the user's locale.

### **Example 22-11. Creating a custom help directory**

The following example changes the base URL to a custom help directory. You must make changes to two files: `com.documentum.help.helpProp.properties` and `/WEB-INF/classes/com/documentum/web/common/HelpService.properties`. Your custom copies of these files will override the base locations.

Create the directory structure `/custom/myhelp` in your application.

Create the directory structure `/custom/strings/com/documentum/web/common` and place in it a copy of the file `HelpService.properties` from `/WEB-INF/classes/com/documentum/web/common`. Open this file and change the entry for the base path to the following:

```
HelpUrl=/custom/myhelp/help.jsp
```

Create the directory structure `/custom/strings/com/documentum/help` and place in it a copy of the file `helpProp_en.properties`. Add a copy for each locale of help in your application, for example, `helpProp_de_DE.properties`. Open the properties files that you have copied and change the base help URL to the following (English example). Make sure that a directory exists and contains localized help files for your application:

```
url=/custom/myhelp/en
```

Restart your application server to see the help files from the new directory.

## Localizing help files

You can add help files for each locale that is supported by your application. The help service locates the localized file based on the user's selected locale. For example, the English help file for the delete component, `deleting.htm`, is located in `/help/en`, and the French file, also named `deleting.htm`, is located in `/help/fr`.

## Utilities

The following utility classes and services are provided by the WDK 5 framework:

- [Clipboard service, page 595](#)
- [Rendering messages to users, page 600](#)
- [Reporting errors, page 601](#)
- [Version utility, page 603](#)
- [Encoding utilities, page 604](#)
- [Input mask, page 606](#)

## Clipboard service

The clipboard service provides a session-based clipboard to handle copy, move, and link operations on multiple objects across components. The user adds items from one

or more folder locations to the clipboard modal window using the Add to Clipboard menu action. The user then navigates to the destination folder for the copy, move, or link operation. The selected copy, move, or link operation is performed on all items in the clipboard. The next Add performed after a copy, move, or link command clears the current contents of the clipboard.

Clipboard operations are verified through the action service. The default action service preconditions are that the user has read permissions on the source object and full write permissions on the destination folder. Items can be added to the clipboard from more than one repository if the user has identical login credentials for the two repositories (same user name and password).

The following topics describe the implementation of clipboard support in a custom component:

- [Clipboard APIs, page 596](#)
- [Using the clipboard in a component, page 597](#)
- [Location and refresh, page 598](#)
- [Clipboard Action Filtering, page 599](#)

## Clipboard APIs

The clipboard service has the following classes and interfaces in the package `com.documentum.web.formext.clipboard`:

**Clipboard** — Implements `IClipboard` and provides add, copy, and paste functions as well as clipboard maintenance functions

**IClipboard** — Returned by `Component.getClipboard()`.

**IClipboardCutHandler** — This interface is called after a successful `pasteAsMove()` clipboard operation. The interface declares a single method: the `onCut()` event handler.

**IClipboardPasteHandler** — This interface is called after a successful `pasteAsCopy()` or `pasteAsLink()` clipboard operation. Declares `onPasteAsCopy()` and `onPasteAsLink()` methods.

**ClipboardUtil** — This class provides a set of static utility methods that are the default implementation of deep copy, deep delete, and link operations on standard `dm_sysobject` and `dm_folder` objects.

The `IClipboard` interface declares the following methods, which must be handled by custom components that use the clipboard:

- `add(String strObjectId, IClipboardCutHandler handler)`: Adds one or more object IDs to the clipboard. The first parameter can be an array of object IDs. The second parameter (`IClipboardCutHandler`) will be called if the item is used for a move operation.
- `pasteAsCopy(String strObjectId, IClipboardPasteHandler handler)`: Pastes one or more object IDs from the clipboard using the specified paste handler. The first parameter can be an array of object IDs. If the `IClipboardPasteHandler` value is null, the current paste handler will be used.
- `pasteAsLink()`: Same as `pasteAsCopy()`. The link operation is performed handled by the paste handler.
- `pasteAsMove()`: Pastes the specified items from the clipboard using the specified paste handler and then cuts the original items from their original location by calling the associated `IClipboardCutHandler` for each item. The first parameter can be an array of object IDs. The second parameter: This handler will be used if a null value is passed for the handler during any future paste operations.

## Using the clipboard in a component

You can enable your components to use the clipboard for Documentum objects by calling `getClipboard()` on the `Component` class to retrieve the `IClipboard` interface for the current user.

Your component must import `IClipboard`, `IClipboardCutHandler`, and `IClipboardPasteHandler` in your component class and implement the two handler interfaces. You can also import the utility class `ClipboardUtil` if your component can use the default implementation of the copy, link, or delete operations.

Your component must implement the following clipboard handler methods:

**`onCut(String strObjectId)`** — Event handler for `onCut` event. For example:

```
public void onCut(String strObjectId)
{
 try
 {
 ClipboardUtil.deleteObject(getDfSession(), strObjectId);
 }
 catch (DfException dfe)
 {
 WebComponentErrorService.getService().setNonFatalError(this,
 "MSG_CLIPBOARD_CUT_ERROR", dfe);
 }
}
```

**`onPasteAsCopy(String strObjectId)`** — Event handler for `onPasteAsCopy` event. For example:

```
public void onPasteAsCopy(String strObjectId)
{
 try
 {
 ClipboardUtil.copyObject(getDfSession(), strObjectId, m_strFolderId);
 }
 catch (DfException dfe)
 {
 WebComponentErrorService.getService().setNonFatalError(
 this, "MSG_CLIPBOARD_PASTE_ERROR", dfe);
 }
}
```

**onPasteAsLink(String strObjectId)** — Event handler for onPasteAsLink event. For example:

```
public void onPasteAsLink(String strObjectId)
{
 try
 {
 ClipboardUtil.linkObject(getDfSession(), strObjectId, m_strFolderId);
 }
 catch (DfException dfe)
 {
 WebComponentErrorService.getService().setNonFatalError(
 this, "MSG_CLIPBOARD_PASTE_ERROR", dfe);
 }
}
```

## Location and refresh

If you handle an operation that requires a location, you must resolve the location for the current component, for example, a folder path for a repository list, or an inbox package object for an inbox component.

In the paste handler examples, the component gets the target folder ID from the FolderUtil utility class `com.documentum.web.formext.docbase.FolderUtil`:

```
m_strFolderId = FolderUtil.getFolderId(strFolderPath);
```

The target folder ID is required for a copy or link operation.

After a clipboard operation, the component that launched the clipboard may need to refresh the display to show changes from the operation. You should override `onRefreshData()` and call `refresh()` on any data provider controls such as data grid. This will reload the data that may have changed.

## Clipboard Action Filtering

You can enable action filtering on clipboard functions using the action service. Action classes in the package `com.documentum.webcomponent.environment.actions` allow you to set preconditions for clipboard actions. For example, the action definition in `dm_sysobjects` is `/webcomponent/config/actionsdm_sysobject__actions.xml` defines the following actions:

- `addtoclipboard`: Requires `objectId` parameter. `lockOwner` and `object type` parameters are optional. Two precondition classes are defined: `AddToClipboardAction` and `RolePrecondition`. The `AddToClipboardAction` class gets a Documentum session, looks up the object type from its ID, gets the lock owner, and allows execution if there is no lock or the current user has the lock.
- `move`: No parameters are defined. The `moveAction` precondition class is defined. The `moveAction queryExecute()` method checks whether there are items in the clipboard. The `moveAction()` method calls `IClipboard pasteAsMove()`.
- `copy`: No parameters are defined. The `copyAction` precondition class is defined. The `copyAction queryExecute()` method checks whether there are items in the clipboard. The `copyAction()` method calls `IClipboard pasteAsCopy()`.
- `link`: No parameters are defined. The `linkAction` precondition class is defined. The `linkAction()` method calls `IClipboard pasteAsLink()`.

To enable a clipboard action filter, reference the action in a button or link. For example, the `cut` action ID is defined in `/webcomponent/config/environment/clipboard/dm_sysobject_clipboard_actions.xml`. In `/webcomponent/navigation/drilldown/drilldown_body.jsp`, an `Add to Clipboard` button is placed on the page:

```
<dmfx:actionlink ... name='addtoclip' ... action='addtoclipboard'>
```

The action `addtoclipboard` is resolved by the configuration service to its definition in `dm_sysobject_clipboard_actions.xml`, and the preconditions are evaluated.

To remove a clipboard action for a specific qualifier such as object type, use the `notdefined` attribute on a scoped action. In the following example, the clipboard actions are disabled for `my_custom_type` objects:

```
<scope type="my_custom_type">
 <action id="addtoclipboard" notdefined="true">
 </action>
</scope>
```

For more information on using the action service, refer to [Chapter 15, Customizing actions](#).

## Rendering messages to users

The message service maintains in the session a list of messages to the user from various services and components. You can use the message service to post success or error messages when you do not need a special UI component such as a prompt.

Some of the most commonly used message service interfaces in `com.documentum.webcomponent.library.messages.MessageService` class are described below:

**addMessage(Form form, String strMessagePropId)** — Adds a message to the form (usually the status bar component). Parameters: Form name, NLS ID of the message string. For example:

```
MessageService.addMessage(this, "MSG_LDAP_CONNECTION_FAILED");
```

An optional third is an array of parameters for NLS string substitution. Each parameter substitutes a placeholder in the message string corresponding to a numbered position in the array. The placeholder takes the form `{n}` where `n` is the number of the parameter starting from zero.

**addDetailedMessage(Form form, String strMessagePropId, Object oParams, String strDetail, boolean bHighPriority)** — Adds a message with additional detail to the message list. The first parameter can be substituted with an NLS resource class (`NlsResourceClass`) instead of a form class. The third parameter is optional (an array of NLS Java token parameters). Refer to the Javadocs for more detail on this overloaded method.

The message is taken from the NLS property file, and the additional detail field supplies information that is not externalized or translated, such as an exception message from an error. Optionally, the user can mark a message as high priority, which are shown in red in the View All Messages screen. For example:

```
MessageService.addDetailedMessage(
 form, strMessagePropId, oParams, exception.getMessage(), true);
```

The Messages component uses the message service to get the result set of messages and initialize a data grid with the result set. For example:

```
Datagrid msgGrid = (Datagrid)getControl(CONTROL_GRID, Datagrid.class);
ResultSet res = MessageService.getResultSet();
msgGrid.getDataProvider().setScrollableResultSet((ScrollableResultSet)res);
```

### To display a message in a component:

1. Insert an entry into the NLS file for the component or form. The following examples show an entry without and with Java token parameters:

```
//without substitution
MY_MESSAGE=That operation cannot be performed in this component.
//with substitution
MY_MESSAGE_2=The operation cannot be performed:{0} Component: {1}
```



2. Import the message service in your behavior class.
3. At the point in your behavior class where the message is posted, call `MessageService.addMessage()`. The following example gets the message strings above:

```
//without substitution
MessageService.addMessage(this, "MY_MESSAGE");
//with substitution

String[] strParams = new String[] {"Delete", "VDM editor"};
MessageService.addMessage(this, "MY_MESSAGE_2", strParams);
```

**Note:** The parameters are an array, so any object can be passed in if it can be converted to a string.

## Reporting errors

Error handling is governed by the error message service. The base class, `ErrorMessageService`, is in the `com.documentum.web.common` package. When a fatal error is thrown, the `WrapperRuntimeException` class calls `ErrorMessageService` and saves the error into the session. The message can then be retrieved by the error message component.

Errors thrown during processing of JSP pages are handled by the error handler servlet, which saves the stack trace in the session.

The error message service is configurable in the application configuration file `app.xml`, so that each application can have its own error message service.

For information on using tracing to track down errors, refer to [Tracing, page 333](#).

**WrapperRuntimeException** — Use `WrapperRuntimeException` to pass specific error messages in your catch blocks. For example:

```
private static String adjustEventName(String eventName)
{
 try
 {
 if(eventName.indexOf(' ') == -1)
 {
 throw new WrapperRuntimeException("no event name", e);
 }
 }
 ...
}
```

or simply:

```
catch (Exception e)
{
 throw new WrapperRuntimeException(e);
}
```

**Reporting user errors** — You can use the `WebComponentErrorService` to report error messages from components to users. In the following example, the error message service reports an error when the user tries to view an object that does not exist. The method `setReturnError()` is in the `Form` class, so it is inherited by all components:

```
catch (Exception e)
{
 setReturnError("MSG_CANNOT_FETCH_OBJECT", null, e);
 WebComponentErrorService.getService().setNonFatalError(
 this, "MSG_CANNOT_FETCH_OBJECT", e);
}
```

Where "this" is the component or form object, and the error string is an NLS key that exists in your component NLS resource file.

If you are handling an error in a contained component, you cannot call `setComponentReturn()`. You could create a custom error page (`<pages>.<errorPage>` in your component definition) to display your error message. The following example sets the error as the value of a label control (name="errorLabel") on your error page before calling the page:

```
try
{
 //code that anticipates an error
}
catch(Exception e)
{
 WebComponentErrorService.getService().setNonFatalError(this, "
 TestMessage", new Exception("Test Exception"));
 Label lblErr = (Label)getControl("errorLabel");
 lblErr.setText("Your error message and instructions go here...");
 setComponentPage("errorPage");
}
```

**Custom error message service** — You can register an error message service for your application layer. Use the `<errormessageservice>` element in `app.xml` for your application layer to specify the class that handles your application error messages. For example, the webcomponent `app.xml` specifies the following:

```
<errormessageservice>
 <class>com.documentum.webcomponent.common.WebComponentErrorService
 </class>
</errormessageservice>
```

Because the error message service class is registered for an application layer, any control or component class can use the same call. For example:

```
catch(DfException exp)
{
 // Writes a message to warn of the error
 ErrorMessageService.getService().setNonFatalError(
 this, "MSG_ERROR_USER_IMPORT", exp);
}
```

The class `ErrorMessageService` provides several methods. One of the methods that is commonly used by component classes is `setNonFatalError()`: Allows an exception to be set as a non-fatal error, along with a message. For example, you can add a detailed exception message from a DFC operation. Parameters:

- `Form`: Form class that contains the context of the error
- `strMessagePropId`: String ID of the error message in the form's property file
- `oParams`: (Optional) Array of parameters for use in the NLS string
- `exception`: The Exception object.

**Error handler servlet** — The error handler servlet calls the error message component and saves the stack trace in the session. If you do not specify an error page in your JSP page, the exception will be displayed inline. Each JSP page in your application should include a directive to redirect to the `errormessage` page in the case of an exception. The syntax for the directive is:

```
<% page errorPage="/wdk/errorhandler.jsp" %>
```

This error message service class is called by default for all components in the webcomponent layer because it is the registered error message handler in the webcomponent `app.xml` file.

The error handler JSP page has a directive at the top of it that specifies to the J2EE server that it is an error page. The servlet is called by the J2EE server when there is an error on a JSP page, and the servlet then opens a new window to load the error message component.

**Error message component** — The error message component gets a `WrapperRuntimeException` from the session, where it was saved by the error handler servlet. The component extracts the original exception and displays the message and stack trace.

## Version utility

The utility class `com.documentum.web.util.Version` provides methods for comparison of version strings. For Documentum versions, the standard version string format is `[(digit)*[letter]].(digit)*[letter]].(digit)*[letter]].(digit)*[letter]]`.

The methods `compareTo(Version ver)` and `compareTo(String str)` compare the input version object or string to the current version string. For example:

```
private static boolean isOldMacOS()
{
 String osname = System.getProperty("os.name");
 String osversion = System.getProperty("os.version");
 return (osname.indexOf("Mac OS") != -1) && ((new Version("10.0")).
 compareTo(osversion) > 0);
}
```

The next example tests whether the repository version is greater than a specified minor version:

```
private static final String MIN_SERVER_VERSION = "5.0";
IDfSession dfsession = getDfSession();
if ((new Version(MIN_SERVER_VERSION).compareTo(
 dfsession.getServerVersion()) <= 0)
 {
 m_bVersionOk = Boolean.TRUE;
 }
}
```

## Encoding utilities

The following encoding utilities are provided in WDK:

- [SafeHTMLString](#), page 604
- [StringUtil](#), page 605
- [ZipArchive](#), page 606

### SafeHTMLString

The class `com.documentum.web.util.SafeHTMLString` prints HTML-safe strings by escaping any embedded characters that would be interpreted as HTML by a browser. The `escape()` method takes a string and returns a safe string. In the following example, event arguments are encoded to be returned as a JavaScript string:

```
public String getOnClickScript ()
{
 StringBuffer buf = new StringBuffer(128);
 buf.append(menu.getEventHandlerMethod("onclick"))
 .append("(this");
 // escape the output params
 ArgumentList eventArgs = getEventArgs();
 if (eventArgs != null)
 {
 Iterator iterNames = eventArgs.nameIterator();
 while (iterNames.hasNext())
 {
 String strArgName = (String) iterNames.next();
 String strArgValue = (String) eventArgs.get(strArgName);
 buf.append(",\"");
 buf.append(SafeHTMLString.escape(strArgValue));
 buf.append("\");");
 }
 }
 buf.append(");");
 return buf.toString();
}
```

Use the `escapeScriptLiteral()` method to encode data that will be rendered as a JavaScript argument. It escapes single quotes, double quotes, backslash, and closing tags.

## StringUtil

The class `com.documentum.web.util.StringUtil` supports the printing of HTML-safe strings by escaping any embedded characters that would be interpreted as HTML by a browser. This utility class provides the following methods:

- `escape(String strText, char character)`

Escapes given character in the specified string. For example:

```
m_out.println("<script>setMessage('" + StringUtil.escape(
 strMessage, '\\') + ": " + percent + " %');</script>");
```

- `unicodeEscape(String str)`

Escapes unicode characters in the input string. For example, processing URL parameters:

```
import java.net.URLEncoder;
...
bufferUrl.append("&");
bufferUrl.append(strArgName).append("=");
if (strArgValue != null && strArgValue.length() > 0)
{
 bufferUrl.append(URLEncoder.encode(
 StringUtil.unicodeEscape(strArgValue)));
}
```

- `replace(String strSource, String strSearch, String strReplace)`

Replaces occurrences of a search string with a replace string. For example:

```
String strDateFormat = df.format(date);
strDateFormat = StringUtil.replace(strDateFormat, "2003", "yyyy");
```

- `splitString(String strParameters, String strDelimiter)`

Splits a token separated string of values into a vector of strings. If no value is supplied at one position, the string must contain the value `null` at that position. For example:

```
strCheckoutPaths = Base64.decode(strCheckoutPaths);
Vector checkoutPaths = StringUtil.splitString(
 strCheckoutPaths, IContentXferConstants.PARAMETER_SEPARATOR);
...
public static final String PARAMETER_SEPARATOR = "|";
```

## ZipArchive

The class `com.documentum.web.util.ZipArchive` creates a zip archive based on a file path in a string. In the following example, the XML files within the current war file are read and added to a `ConfigFile` object:

```
String strWARFile = System.getProperty(strAppName + ".war");
ZipArchive warArchive = null;
try
{
 warArchive = new ZipArchive(strWARFile);

 //Get all files inside given folder recursively (true = recurse)
 Iterator iterSubs = warArchive.listFilesInFolder(strFolderPath, true);
 while (iterSubs.hasNext())
 {
 String strFilePathName = (String)iterSubs.next();
 if (strFilePathName.endsWith(".xml"))
 {
 ConfigFile configFile = new ConfigFile(strFilePathName, strAppName);
 configFile.add(configFile);
 }
 }
}
```

Be sure to close the archive when you are finished with it:

```
finally
{
 warArchive.close();
}
```

## Input mask

The `InputMaskUtil` class implements an input mask that accepts a string of characters. If you need to mask user input, you can also use the input mask control. Refer to *Web Development Kit Reference Guide* for information on the input mask control.

To use a mask, implement a method that calls `InputMaskUtil.parseMaskString()`. This method will flag the escaped characters in your mask string. Then you call `InputMaskUtil.isCharValid()` on the input string. The following example uses an input mask to validate a string that is passed in:

```
protected boolean doCompare(String strValue)
{
 boolean bValid = false;
 // Retrieve mask to validate against
 String strMask = getMask();
 int valueLen = strValue.indexOf(0);
 valueLen = strValue.length();
 // if the input value is shorter than the mask, the value is not valid.
```

```
if (valueLen < strMask.length())
{
 bValid = false;
}
else
{
 bValid = true; // check the strValue in the for loop
 for (int j = 0; j<valueLen; j++)
 {
 if (!InputMaskUtil.isCharValid(j, strValue.charAt(j),
 strMask, false))
 {
 bValid = false;
 break;
 }
 }
 return bValid;
}
}
```





# Using Business Objects in WDK

Documentum business objects (DBOs) are Java components that use DFC to perform business logic, independent of the presentation layer or application logic. A business object can perform the same operation for a Web application and a Desktop application.

WDK installs several business objects: workflow inbox, workflow reporting, subscriptions, and content intelligence services. Web Publisher uses a few dozen business objects. The Documentum developer Web site ([developer.documentum.com](http://developer.documentum.com)) provides many other examples of useful business objects, such as autonumbering and autonaming, deep export, zip service, and recycle bin.

In the Web environment, DBOs can be included on the application server. On the Content Server, the DBOs can exist as server methods, jobs and lifecycle procedures. You can not directly call a DBO on the Content Server from a client such as WDK.

DBOs are written in Java. You then instantiate the DBO and call its methods in a WDK client.

The following topics describe how to call DBOs from a WDK-based application:

- [Calling an SBO method, page 609](#)
- [Using TBOs, page 610](#)

For information on how to create DBOs, refer to the *Documentum Foundation Classes Development Guide*.

## Calling an SBO method

A service-based business object (SBO) provides functionality as a service to be used by applications. The SBO is not specific to a particular object type. Examples of SBOs include subscriptions and inbox.

### To call an SBO method:

1. Use the `SessionManagerHttpBinding` to create a Session Manager object.
2. Use the `IDfClient.newService()` factory method, passing the service name and Session Manager, to create the SBO instance.

3. Set the repository for the service. The repository name must be passed either to the SBO or to every method of the SBO.
4. Call methods from the SBO.

**Example 23-1. Calling an SBO method**

The following example from `DRLInboxItemViewtAction` instantiates the inbox service and calls a service method:

```
final IDfSessionManager manager = SessionManagerHttpBinding.getSessionManager();
IInbox inboxService = (IInbox)DfClient.getLocalClient().newService(
 IInbox.class.getName(), manager);
inboxService.setDocbase(SessionManagerHttpBinding.getCurrentDocbase());

IDfSession session = null;
try
{
 session = manager.getSession(SessionManagerHttpBinding.getCurrentDocbase());
 String strObjectId = getTaskId(args, session);
 args.replace("objectId", strObjectId);
 context.set("objectId", strObjectId);

 //call the service method
 ITask task = inboxService.getTask(new DfId(strObjectId), true);
 ...
}
```

## Using TBOs

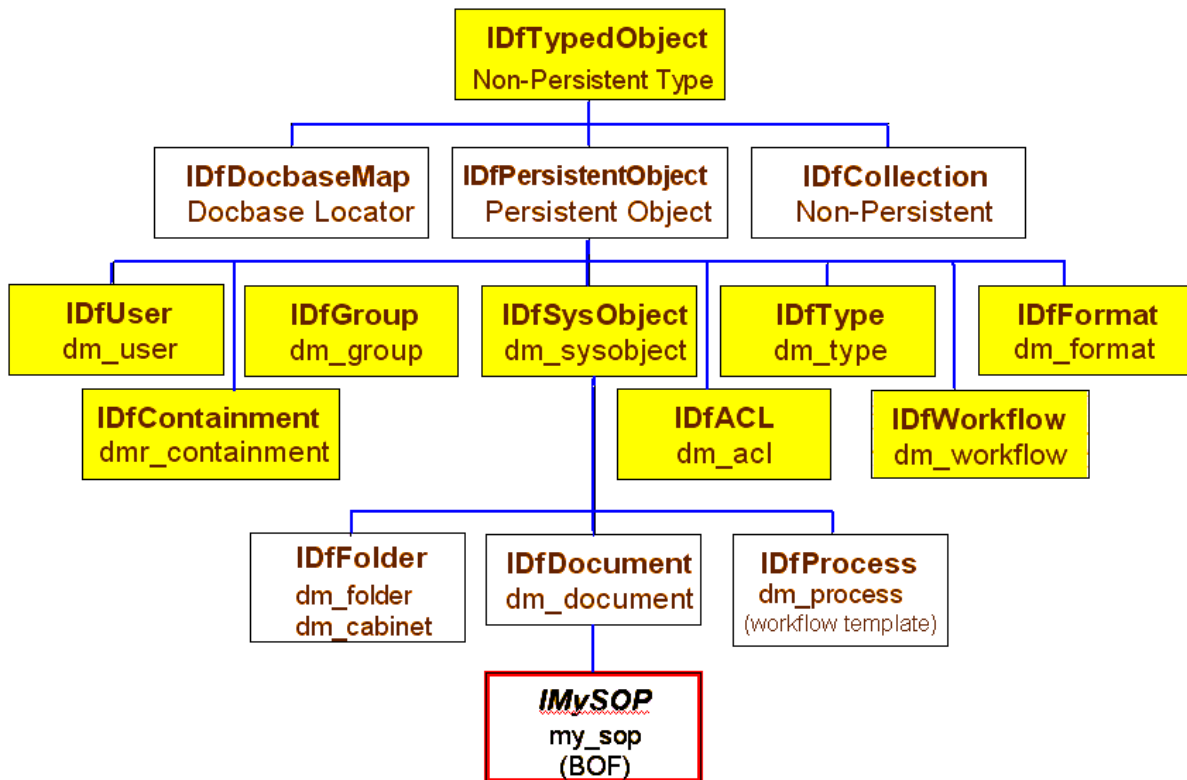
A type-based business object (TBO) allows you to override DFC methods for a specific object type. Once your TBO is registered, your TBO methods will be called when the relevant operation is called on the TBO type. For example, your TBO for `my_sop` type overrides the `checkin` operation and creates a rendition on `checkin`. When the user checks in a document of the type `my_sop`, your `checkin` operation is performed.

TBOs are instantiated when a user performs an action on a custom type. Unlike SBOs, you do not need to explicitly instantiate TBOs. TBOs can be called only for custom server object types and not for Documentum types such as `dm_document` or `dm_sysobject`.

Use a TBO to override methods in the base class or create custom methods for the type. TBOs extend `DfPersistentObject` or subtypes. For example, you could override `DfFolder` or `DfDocument` behavior in your TBO. The TBO interface must extend `IDfBusinessObject`.

The Documentum object hierarchy with a type-based object is shown below:

Figure 23-1. Documentum object hierarchy



When you create a TBO and register it with the DBOR registry, the operations in your TBO are automatically performed for objects of that type in your WDK-based applications. Refer to *Documentum Foundation Classes Development Guide* for information on creating TBOs.



# Customization Examples

WDK client applications such as Webtop, Web Publisher, Digital Asset Manager, or Documentation Administrator can be configured and customized using the general guidelines in this guide. For information on locating the correct files for configuration, refer to [Finding files to configure controls](#), page 162.

This section describes customization of features that are specific to a client application:

- [Displaying Objects: Datagrid and objectgrid](#), page 613
- [Creating a component](#), page 614
- [Customizing components](#), page 617
- [Customizing controls](#), page 623
- [Customizing actions](#), page 625
- [Custom queries and data sources](#), page 630
- [Creating a validator](#), page 633
- [Creating a qualifier](#), page 635
- [Using a prompt within a container](#), page 636

## Displaying Objects: Datagrid and objectgrid

The `objectgrid` component and the `datagrid` control can both display objects.

The `objectgrid` component renders attributes for objects returned by a query on `dm_sysobjects`. The component gets a data provider and sets the list of attributes to be displayed in a `datagrid`. The objects are displayed in `objectgrid.jsp`. Some examples of components that extend `objectgrid` to display objects are `locations`, `relationships`, `renditions`, `versions`, and `history`.

Some components display objects using a `datagrid` only. These component generally display lists of objects that are not `dm_sysobjects`. A component that displays a `datagrid` must get a data provider for the `datagrid` and provide the query to the data provider, just as the `objectgrid` component does. For example, the `formatlist` component displays

all formats in the repository. The formatlist component gets the data provider in its `onInit()` method as follows:

```
datagrid.getDataProvider().setDfSession(getDfSession());
```

The formatlist component then sets the query:

```
m_strQuery = "select r_object_id,name,description,
dos_extension,com_class_id from dm_format";
datagrid.getDataProvider().setQuery(m_strQuery);
```

## Creating a component

Your component can extend another component, or you can create a new component from scratch. Your component must have the following parts:

- A component class that extends `Component` or one of its subclasses and implements your component behavior
- Getter and setter methods in the component class to access public component properties
- A component configuration file
- One or more JSP pages
- Means of launching the component: Menu, button, or link

The following steps outline the requirements for creating a component. For a detailed example of creating a component, refer to the *Web Development Kit and Applications Tutorial*.

1. [Extending a component, page 614](#)
2. [Creating a component definition, page 615](#)
3. [Adding component parameters to the component class, page 615](#)
4. [Getting data, page 616](#)
5. [Creating the component JSP pages, page 616](#)
6. [Implementing navigation in a component, page 616](#)

## Extending a component

You can take advantage of functionality in a component by extending it. For example, the Webtop advanced search component extends the WDK search component. The component class extends the parent component class, and the XML definition extends

the parent component definition. The following procedures lists the steps that were required to extend the search component.

For more information on extending a component definition, refer to [Component inheritance \(extends\)](#), page 224.

## Creating a component definition

Your component configuration file must contain certain elements. For more information on the elements in a component configuration file, refer to [Component configuration file](#), page 221:

Define the required and optional parameters in your component definition. Values for these parameters will be passed to the component class in the Request. A parameter definition has the following syntax:

```
<param name="packageName" required="true"></param>
```

Because you have specified that this parameter is required, you can get it from the argument list that is passed to your component `onInit()` method when the component is called:

```
public void onInit(ArgumentList args)
{
 String strFolderPath = args.get("packageName");
 ...
}
```

## Adding component parameters to the component class

The component's settings are defined in the definition XML file by the `<params>` tag. The parameters are passed to the component when it is first called. The component class will enforce the use of required parameters.

### Example 24-1. Adding parameters

User-defined arguments may be added to the component parameters. In the following example, an extended attributes component specifies an "approved" parameter in addition to the two parameters in the base component:

```
<params>
 <param name="objectId" required="true"></param>
 <param name="readOnly" required="false"></param>
 <param name="approved" required="false"></param>
</params>
```

The parameters are passed by the component dispatcher to the attributes component. The extended attributes component class then gets the parameters in the `onInit()` event handler method ( the event handler for the component initialization lifecycle event):

```
public void onInit(ArgumentList args)
{
 super.onInit(args);
 String strObjectId = args.get("objectId");
 String strApproved = args.get("approved");
 if (strApproved.equalsIgnoreCase("true"))
 {
 bApproved = true;
 }
}
```

## Getting data

Your component can get data using a data source such as a JDBC result set, an XML result set, or a DFC session interface. Refer to [Getting data, page 390](#) for examples.

## Creating the component JSP pages

Your component should have at least one JSP page. The initial component JSP page is registered in the component definition as the value of the `<pages>.<start>` element.

The JSP pages in your component, if they are not included in another JSP page, should have the `<dmf:webform/>` tag somewhere before the HTML `<body>` tag. Immediately after the `<body>` tag, add `<dmf:form>` to include all of the HTML, JavaScript, and JSP tag content of your page. Close the form tag just before you close the `<body>` tag.

Your component class must handle all events fired on the JSP page, unless they are handled in client-side JavaScript on the page.

Refer to [Contents of a WDK JSP page, page 234](#) for an overview of component JSP pages.

## Implementing navigation in a component

You can implement navigation between your component pages and navigation to other components using the component navigation methods:

- `setComponentPage(String strPageName)`: Navigates to the specified component page URL as defined by the logical page name in the component definition XML file.



- `setComponentJump(String strComponentName, String strStartPage, Arguments args, Context context)`: Jumps to the component specified by the component name.
- `setComponentNest(String strComponentName, String strStartPage, Arguments args, Context context)`: Nests to the component specified by the component name.
- `setComponentReturn()`: Returns from a nested call.

Refer to [Calling a container from a server class, page 457](#) for more information.

## Customizing components

The following examples describe common customizations of components:

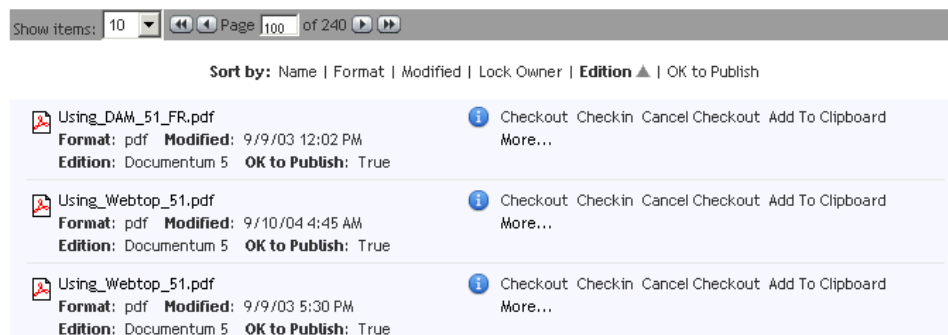
- [Displaying a single custom object type \(object grid\), page 617](#)
- [Getting a component reference in a JSP page, page 623](#)

### Displaying a single custom object type (object grid)

The drilldown component allows you to display all objects in a selected cabinet or folder. You may want to display only a selected object type. For this purpose, use a component that extends the `objectgrid` component.

The following example creates a component that displays a custom object type, technical publications web. It displays two custom attributes for the type and allows you to sort on those attributes, Edition and Publish:

**Figure 24-1. Custom type object grid**



The following topics describe how to create a custom object grid:

- [Creating the custom grid component definition, page 618](#)
- [Creating the object grid class, page 619](#)

- [Adding custom columns to the display, page 619](#)
- [Adding externalized strings, page 620](#)
- [Launching the object grid component, page 621](#)

For the working code samples, refer to the Documentum Developer Web site (<http://developer.documentum.com>).

## Creating the custom grid component definition

Create a component configuration file, for example, `webdocs_component.xml`, in `/custom/config` to define your new component. This definition is a copy of the `objectgrid` component definition that specifies a new component JSP page, new component class, new properties file. At the end of the columns list, two new columns have been added to display custom attributes:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<config version='1.0'>
 <scope>
 <component id="webdocs">
 <pages>
 <start>/custom/webdocs/webdocs.jsp</start>
 </pages>
 <class>com.mycompany.webdocs.WebDocs</class>
 <nlsbundle>com.mycompany.webdocs.WebDocsNlsProp</nlsbundle>
 <header visible='false' />
 <columns>
 ...
 <column>
 <attribute>r_lock_owner</attribute>
 <label><nlsid>MSG_LOCK_OWNER</nlsid></label>
 <visible>true</visible>
 </column>
 <column>
 <attribute>tp_edition</attribute>
 <label>Edition</label>
 <visible>true</visible>
 </column><column>
 <attribute>tp_web_viewable</attribute>
 <label>Publish?</label>
 <visible>true</visible>
 </column>
 </columns>
 </component>
 </scope>
</config>
```

**Note:** When you add columns to the component definition, you must also add them to the JSP page in order for them to be displayed.

## Creating the object grid class

You specified a component class in the webdocs component definition. The objectgrid component class is abstract and should be extended. Create the custom object grid class file in your J2EE IDE and instruct your IDE to compile the class file in `/WEB-INF/classes/com/company_name/class_name`. This class simply overrides the query that supplies the data for the object grid. The internal attributes are supplied by the query string defined in ObjectGrid: `r_object_id`, `object_name`, `r_link_cnt`, `r_is_virtual_doc`, `owner_name`, `r_object_type`, `a_content_type`, `r_lock_owner`, `r_content_size`, `i_is_reference`. The visible attributes are supplied by the list of columns in the component definition.

The object grid class should override the query similar to the following example. The FROM statement specifies the object type.

```
package com.mycompany.webdocs;
import com.documentum.web.common.ArgumentList;
import com.documentum.webcomponent.navigation.objectgrid.ObjectGrid;

public class WebDocs extends ObjectGrid
{
 public void onInit(ArgumentList args)
 {
 super.onInit(args);
 }

 /**
 * Supplies the query for the component.
 * @param strVisibleAttrs visible attributes list
 * @param args Argument list
 * @return String the DQL statement for component.
 */
 protected String getQuery(String strVisibleAttrs, ArgumentList args)
 {
 StringBuffer strQueryBuf = new StringBuffer(512);
 strQueryBuf.append("SELECT DISTINCT r_object_id as sortby,")
 .append(strVisibleAttrs).append(INTERNAL_ATTRS)
 .append(" FROM technical_publications_web ORDER BY object_name");

 return strQueryBuf.toString();
 }
}
```

Restart the application server to pick up your new component class and definition.

## Adding custom columns to the display

The objectgrid component JSP page `objectgrid.jsp`, located in `/webcomponent/navigation/objectgrid`, can display the result sets of `dm_sysobject` queries only. It can display the following attributes of `dm_sysobject` type: `object_name`, `r_version_label`, `r_content_size`,

r\_modify\_date, a\_content\_type, title, authors, owner\_name, r\_lock\_owner , r\_object\_type, r\_creation\_date, r\_modifier, r\_access\_date, group\_name, r\_creator\_name, and primary folder path. The implementing component must provide its own JSP pages for non-dm\_sysobject type queries.

Custom attributes can be added to the component definition in preparation for formatting and display. These attributes must also be added to the component layout JSP for display. Refer to [Creating the custom grid component definition, page 618](#) for an example.

To display the custom attributes using a celltemplate control, you can either specify the field in the celltemplate tag or specify some generic data type. The following example adds a custom field to the column header block. The celltemplate tag attribute "field" specifies the attribute to be displayed, and the datafield attribute on the datasortlink tag specifies the label for the column header:

```
<dmf:celltemplate field='tp_edition'>
 <dmf:datasortlink cssclass='drilldownFileInfo' name='sort5'
 datafield='tp_edition' reversesort='true' />
</dmf:celltemplate>
```

You also add your columns to the dmf:datagridRow control. The following example adds the tp\_edition column:

```
<dmf:celltemplate field='tp_edition'>
 <dmf:label cssclass='drilldownLabel' />:
 <dmf:label datafield='tp_edition' />
</dmf:celltemplate>
```

You may also need to pass attribute values from your objects to actions if the action, or the component called by the action, uses these values. (You can find this out by examining the parameters in the action or component definition.) This data is provided in the form of argument tags in the actionlinklist tag. In the following example, two custom attribute are passed as arguments:

```
<dmfx:actionlinklist name="<%=ObjectGrid.ACTIONS_LIST%>">
<!-- arguments passed to ALL actions in the list -->
 <dmf:argument name="objectId" datafield="r_object_id"/>
 ...
 <dmf:argument name="tp_edition" datafield="tp_edition"/>
 <dmf:argument name="tp_web_viewable" datafield="tp_web_viewable"/>
</dmfx:actionlinklist>
```

## Adding externalized strings

Your component definition specifies a properties file to display your component strings. Strings are externalized to a properties file to facilitate configuration and localization.

These strings may be used for column headers, such as in your <columns> element in the definition:

```
<column>
 <attribute>tp_edition</attribute>
 <label><nlsid>MSG_EDITION</nlsid>nlsid<</label>
 <visible>>true</visible>
</column>
```

You may also use externalized strings in your JSP page, such as the title:

```
<dmf:label cssclass='drilldownTitle' nlsid='MSG_TITLE' />
```

Create a .properties file in the same directory as your compiled class file, for example, /WEB-INF/classes/com/mycompany/webdocs. You must import the objectgrid component properties file to get the relevant strings, then add your custom strings, for example:

```
NLS_INCLUDES=com.documentum.webcomponent.GenericActionNlsProp,
com.documentum.webcomponent.GenericObjectNlsProp,
com.documentum.webcomponent.navigation.objectgrid.ObjectGridNlsProp

MSG_TITLE=Web Documents
MSG_EDITION=Edition
MSG_PUBLISH=OK to Publish
```

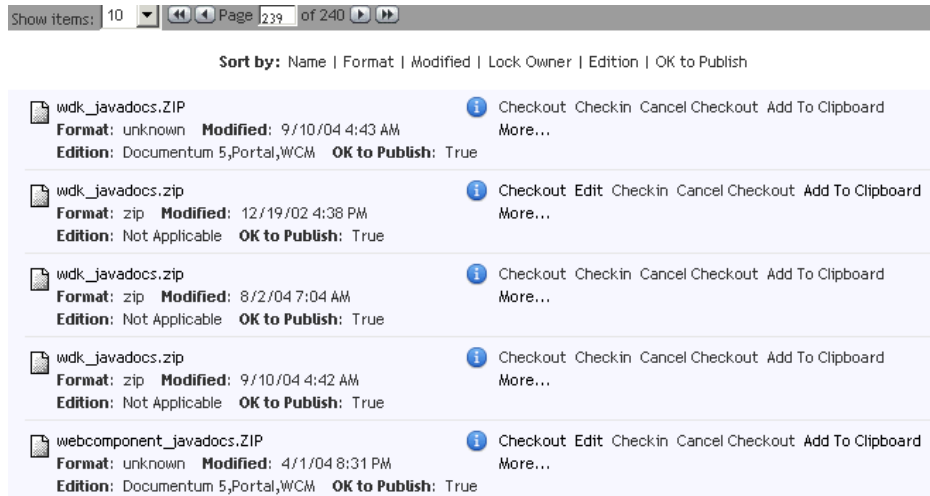
## Launching the object grid component

For testing purposes, you can launch your component with a simple URL, for example:

```
http://localhost/webtop/component/webdocs
```

Your component will fill the entire application frame, similar to the following:

**Figure 24-2. Custom grid launched by URL**

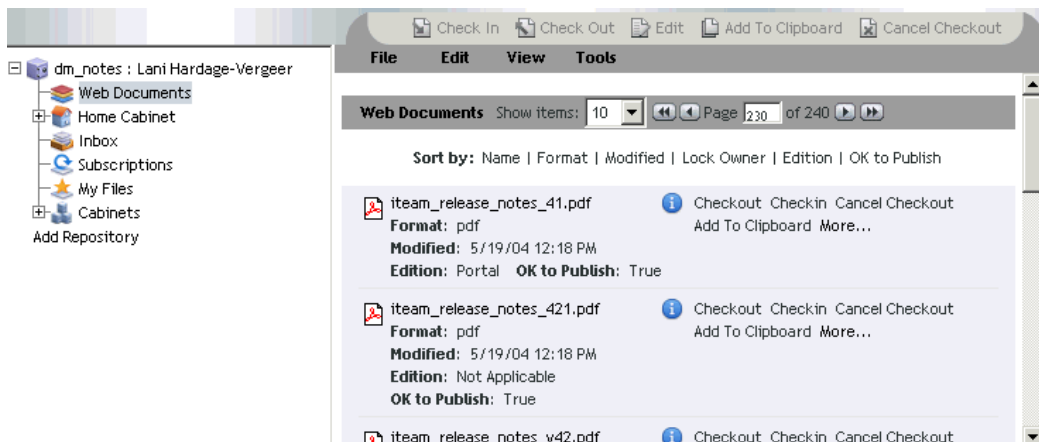


You can add the custom grid to the Webtop browsertree navigation component by extending the browsertree definition and adding a node as follows. Replace the icon file name with an icon that you have placed in `/custom/theme/theme_name/icons`:

```
<docbasenodes>
 <node componentid='webdocs'>
 <icon>classify.gif</icon>
 <label>Web Documents</label>
 <streamlinecomponent>webdocs</streamlinecomponent>
 </node>
```

After you refresh the configurations in memory by navigating to `/wek/refresh.jsp`, the node will be displayed in the tree and will launch your component:

**Figure 24-3. Launching from the Webtop browser tree**



**Note:** The object grid display page, `objectgrid.jsp`, is designed to display a streamline (drilldown) view. To support multiple selection in a classic (objectlist) view, you must design a second JSP page that includes checkboxes.

## Getting a component reference in a JSP page

The component instance is available in the component JSP pages by calling the `pageContext.getAttribute()` method. In the following example, a JSP script gets the component, which is the top-level Form instance, and then calls a component method:

```
<%@ page import="com.documentum.web.formext.component.Component,
 com.documentum.web.form.IParams"%>
<%
 ObjectLocator locatorComp = (ObjectLocator)pageContext.
 getAttribute(IParams.FORM, PageContext.REQUEST_SCOPE);
 if(locatorComp.getTopForm() instanceof LocatorContainer)
 {
 locatorComp.doSomeStuff();
 }
%>
```

## Customizing controls

A control class must extend one of the following abstract control classes:

- `com.documentum.web.form.Control`
- `com.documentum.web.form.control.StringInputControl`
- `com.documentum.web.form.control.BooleanInputControl`

You must also implement a control tag class that extends one of the following tag classes:

- `com.documentum.web.form.ControlTag`
- `com.documentum.web.form.BodyControlTag`
- `com.documentum.web.form.control.StringInputControlTag`
- `com.documentum.web.form.control.BooleanInputControlTag`

The control class must perform the following functions:

- Constructor must accept a container and a name
- Class must override `getEventNames()`
- Class must declare property access methods
- Class must override `updateStateFromRequest()`. When you implement `updateStateFromRequest()`, use `getRequestParameter()` to retrieve a value from the request. This method will encode the value in UTF-8.

## Choosing a control superclass

The following table can help you select a superclass for your control:

**Table 24-1. Choosing a control superclass**

Criterion	Superclasses
Is the control simple, with no user input and no contained tags? (Example: A button that fires events in response to user action)	Use Control and ControlTag. Override renderStart() and renderEnd() to write the HTML rendition.
Does the control accept no user input but contain tags within it? (Example: A panel that displays the controls that it contains)	Use Control and BodyControlTag. Override methods inherited from javax.servlet.jsp.tagext.BodyTagSupport to write the HTML rendition.
Does the control accept a string from user input but contain no tags? (Example: A text box that reads a string value from user input)	Use StringInputControl and StringInputControlTag. Override renderStart() and renderEnd() to write the HTML rendition.
Does the control accept boolean input from a user but contain no tags? (Example: A check box)	Use BooleanInputControl and BooleanInputControlTag. Override renderStart() and renderEnd() to write the HTML rendition.

## Adding control events

Event handlers for control events are specified in the control by calling Control.setEventHandler(). This method registers the named control to handle particular events that are posted to from the client to the server. (Events are not fired on the client.) This method has three parameters:

- Event name: Name of the event
- Event handler method name: Name of the method that handles the event
- Handler: Control object that exposes the event handler method. Null if the form class exposes the event handler method.

The following example in the class MyControl sets an event handler for the onSelect event:

```
public MyClass()
{
 setEventHandler(EVENT_ONSELECT, "onSelect", this);
}
```



```
 final static public String EVENT_ONSELECT = "onselect";
}
```

The server event is called by JavaScript that is rendered by the tag class. The `renderEventArg()` method, available to every tag class, specifies the following parameters:

- Buffer to write to (first parameter),
- HTML event name (optional)
- Server event handler

If only two parameters are passed, the second parameter is the server event. The first example below fires a server event, and the second examples includes the HTML event on the control:

```
protected void renderStart(JspWriter out)
{
 StringBuffer buf = new StringBuffer(192);
 ...
 renderEventArg(buf, MyControl.EVENT_ONSELECT);
}

protected void renderStart(JspWriter out)
{
 StringBuffer buf = new StringBuffer(192);
 ...
 renderEventArg(buf, "onchange", MyControl.EVENT_ONSELECT);
}
```

## Customizing actions

The following topics describe some common action customizations:

- [Adding a custom action, page 625](#)
- [Implementing the action execution class, page 627](#)
- [Custom action execution class with pre- and post-processing, page 629](#)

## Adding a custom action

Your component must perform the following steps to enable a custom action:

1. Add an action-enabled item in the UI. For example:

```
<dmfx:actionmenuitem dynamic='multiselect' name=
 'mypromotelifecycle' nlsid='MSG_PROMOTE_LIFECYCLE'
 action='mypromote' showifinvalid='true'/>
```

2. Create an action definition whose action ID matches the action ID of your action-enabled operation. For example, to match the above example, your action

ID would be `<action id="mypromotelifecycle">`. Specify the following items in the definition:

- Required and optional action parameters
- Optional precondition class with any precondition parameters (user-defined element names and values)
- Execution class with any execution parameters (user-defined element names and values)

For example:

```
<action id="myCheckin">
 <params>
 <param name="objectId" required="true"></param>
 <param name="lockOwner" required="false"></param>
 <param name="ownerName" required="false"></param>
 </params>
 <preconditions>
 <precondition class=
 com.documentum.web.formext.action.RolePrecondition">
 <role>Contributor</role>
 </precondition>
 <precondition class="com.acme.MyCheckinAction">
 </precondition>
 </preconditions>
 <execution class="com.documentum.web.formext.action.LaunchComponent">
 <component>checkin</component>
 <container>checkincontainer</container>
 </execution>
</action>
```

3. Create your precondition class and implement the methods `queryExecute()` and `getRequiredParams()`. These methods are executed when the action control tag is processed on the server. Your method `queryExecute()` method should test for preconditions using the required preconditions. For example (without error-handling code):

```
public boolean queryExecute(String strAction, IConfigElement config,
 ArgumentList arg, Context context, Component component)
{
 // determine whether lock owner matches user
 boolean bExecute = false;
 IDfSession dfSession = component.getDfSession();
 try
 {
 // get lock owner
 String strLockOwner = arg.get("lockOwner");
 // get user
 String strUserName = dfSession.getLoginUserName();

 // compare
 if (strUserName != null && strLockOwner != null &&
 strUserName.equals(strLockOwner))
 {
```

```

 bExecute = true;
 }
}
...
return bExecute;
}

```

4. In your action class, implement the methods `execute()` and `getRequiredParams()` to perform the actual operation. In the example above, the execution is passed to the component class because `LaunchComponent` is specified as the execution class in the action definition. Refer to the next section for examples of an action `execute()` method. ([Implementing the action execution class, page 627](#)).

## Implementing the action execution class

In the following example, the action class creates a URL for a selected object and calls another component to mail the URL to a user. For the full working code in this example, refer to the Documentum developer Web site, <http://developer.documentum.com>.

To implement the action, you must implement the methods `execute()` and `getRequiredParams()`. In the following example, arguments are defined in the action configuration file as follows:

```

<action id="formwebtopurl">
 <params>
 <param name="objectId" required="true"></param>
 <param name="objectId" required="true"></param>
 <param name="objectId" required="true"></param>
 </params>
 <execution class="com.documentum.custom.action.WebtopURLCreator"/>
</action>

```

The JSP page contains a datagrid whose query returns several attributes and objects. Within a `datagridrow` control, the `actionlink` control is generated for each object in the datagrid. The `actionlink` control passes the required parameters as arguments to an `actionlink` control, as follows:

```

<dmfx:actionlink name="createurl" datafield="object_name"
 action="myaction">
 <dmf:argument name="objectId" datafield="r_object_id"/>
 <dmf:argument name="actionId" value="view"/>
 <dmf:argument name="jumpComponent" value="mailer"/>
</dmfx:actionlink>

```

The arguments for the selected object will be passed to the action `execute()` method as an `ArgumentList` object. The component named in the `jumpComponent` argument will be launched by the action.

The `getRequiredParams()` method returns the names of the parameters that are required by the action execution. For example:

```
public String [] getRequiredParams()
{
 return new String [] {"objectId"};
}
```

The `execute()` method accepts the argument values and performs the action. For example:

```
public boolean execute(String strAction, IConfigElement config,
 ArgumentList args, Context context, Component component,
 java.util.Map map)
{
 String webtopUrl = ""; //the complete URL
 String appBaseUrl = "";
 String objectId = "";
 String actionId = "";
 String appHost = "";
 String jumpComponent = "";

 jumpComponent = args.get("jumpComponent");
 PageContext pageContext = component.getPageContext();
 ServletRequest servletRequest = pageContext.getRequest();
 appHost = servletRequest.getServerName() + ":" +
 servletRequest.getServerPort();
 appBaseUrl = component.getBaseUrl();
 objectId = args.get("objectId");
 actionId = args.get("actionId");
 webtopUrl = "http://" + appHost + appBaseUrl + "/action/" +
 actionId + "?objectId=" + objectId;

 ArgumentList mailArgs = new ArgumentList();
 mailArgs.add("mailBody", webtopUrl);
 mailArgs.add("objectId", objectId);
 mailArgs.add("component", jumpComponent);
 component.setComponentNested(
 "dialogcontainer", mailArgs, component.getContext(), null);
 return true;
}
```

## Action tracing

**Turning on action tracing** — Set the following flag in `/WEB-INF/classes/com.documentum.debug.TraceProp.properties` to turn on action service tracing:

```
com.documentum.web.formext.Trace.ACTIONSERVICE=true
```

You can also set this trace flag by navigating to `http://application_context_name/wdk/tracing.jsp`.

The `CONFIGSERVICE` tracing flag is also useful in debugging actions.

## Custom action execution class with pre- and post-processing

Following is a description of the syntax to create an action execution class that performs pre- and post-processing.

1. Extend an existing action execution class. For example:

```
public class CustomActionExecutionClass extends
 SomeExistingActionExecutionClass
```

2. Add your pre-processing code to the beginning of the execute method. For example:

```
{
 public boolean execute(String strAction, IConfigElement config,
 ArgumentList args, final Context context,
 Component component, Map completionArgs)
 {
 // DO PRE ACTION PROCESSING HERE
 }
}
```

3. Implement an action complete listener class. For example:

```
class ActionListener implements IActionCompleteListener
{
 public ActionListener(IActionCompleteListener listener)
 {
 m_listener = listener;
 }
 private IActionCompleteListener m_listener;
}
```

4. Add your post-processing to the listener onComplete() implementation. For example:

```
public void onComplete(String strAction, boolean bSuccess, Map map)
{
 if (m_listener != null)
 {
 m_listener.onComplete(strAction, bSuccess, map);
 }
 // DO POST ACTION PROCESSING HERE
}
```

5. Finish up the execute() method by replacing the listener with any other existing action listener so that it also gets called. For example:

```
// replace complete listener with new one.
IActionCompleteListener completeListener = (
 IActionCompleteListener)completionArgs.get(
 ActionService.COMPLETE_LISTENER)

completionArgs.put(
 ActionService.COMPLETE_LISTENER, new ActionListener(
 completeListener));
```

```
}
```

The complete pseudocode is as follows:

```
public class CustomActionExecutionClass extends
 SomeExistingActionExecutionClass
{
 public boolean execute(String strAction, IConfigElement config,
 ArgumentList args, final Context context,
 Component component, Map completionArgs)
 {
 // DO PRE ACTION PROCESSING HERE
 class ActionListener implements IActionCompleteListener
 {
 public ActionListener(IActionCompleteListener listener)
 {
 m_listener = listener;
 }

 public void onComplete(
 String strAction, boolean bSuccess, Map map)
 {
 if (m_listener != null)
 {
 m_listener.onComplete(strAction, bSuccess, map);
 }

 // DO POST ACTION PROCESSING HERE
 }
 private IActionCompleteListener m_listener;
 }

 // replace complete listener with new one.
 IActionCompleteListener completeListener = (
 IActionCompleteListener) completionArgs.get(
 ActionService.COMPLETE_LISTENER)

 completionArgs.put(ActionService.COMPLETE_LISTENER, new ActionListener(
 completeListener));
 }
}
```

## Custom queries and data sources

The following topics describe how to implement custom queries and data sources:

- [Adding a custom query or data source, page 631](#)
- [Populating a dropdown list with a query, page 632](#)

## Adding a custom query or data source

You can display data from the following sources:

- A hard-coded query string
- A generated query string
- A result set
- A JDBC driver
- An in-memory array

### To display data from a query string

1. Add the following lines to your component behavior class by overriding the `onRender()` lifecycle method:

```
Datagrid myGrid=(Datagrid)getControl("mygrid", Datagrid.class);
myGrid.getDataProvider().setConnection(conn);
```

2. Add the following lines to your component JSP layout page. Modify the query attribute to contain your query. Add controls to the `datagridRow` tag as appropriate, and set the `datafield` attributes of those controls to the columns you have queried and wish to display:

```
<dmf:datagrid name="myGrid" query="select object_name from...">
 <dmf:datagridRow>
 <td><dmf:label datafield="object_name"/></td>
 </dmf:datagridRow>
</dmf:datagrid>
```

### To display data from a generated query string

1. Use the same `onRender()` override as above for a hard-coded query string.
2. Use the same JSP as above, but remove the query attribute from the `datagrid` tag.
3. Place your query string in your component behavior class, overriding the `onInit()` lifecycle method. For example:

```
Datagrid myGrid=(Datagrid)getControl("mygrid", Datagrid.class);
myGrid.getDataProvider().setQuery(myQueryString);
```

If you regenerate the query string, use the same code to re-set the query string whenever it changes.

### To display data from another data source `ResultSet`

1. Use the JSP page as above for a hard-coded query string, but remove the query attribute.
2. Use the following code in your component `onInit()` lifecycle method to override the default behavior:

```
Datagrid myGrid=(Datagrid)getControl("mygrid", Datagrid.class);
myGrid.getDataProvider().setResultSet(myResultSet);
```

3. Re-query your ResultSet and reapply it to the datagrid as appropriate for your data.

**To display a list of data using a JDBC driver** — Use any of the code above, but use your specific Connection object in the setConnection() call in your component onRender() method.

**Note:** For some JDBC sources, the result set may contain column names in all capital letters, for example, Oracle.

**To display a list of data from an in-memory array**

1. Use the JSP page as above for a hard-coded query string, but remove the query attribute.
2. In your custom component, override the onInit() lifecycle method similar to the following example:

```
Datagrid myGrid=(Datagrid) getControl("mygrid", Datagrid.class);
String[] myColumnNames = newString[]{"object_name", "r_object_id"};
TableResultSet mySet = new TableResultSet(myListOfData, myColumnNames);
myGrid.getDataProvider().setScrollableResultSet(mySet);
```

The ArrayList myListOfData[] should be filled with an array containing the data you wish to display. You can also use a vector for this parameter. The String array myColumnNames[] should be filled with the names of columns that you wish to use for your data.

3. Generate the data and reapply the ScrollableResultSet to the datagrid as appropriate for your data. (Refer to example in step 2.)

The WDK scrollable result sets include classes to handle data in arrays, lists, and lists containing arrays (tables).

## Populating a dropdown list with a query

You can use a DQL query to set the results for a dropdown list. The following example from a component class gets a list of formats from the repository and displays them for user selection in a dropdown list control:

```
public void onInit(ArgumentList args)
{
 super.onInit(args);

 // set the session on the data provider control (dropdown list)
 DataDropDownList dropDownList = ((DataDropDownList) getControl(
 FORMAT_RENDITION_LIST, DataDropDownList.class));
 dropDownList.getDataProvider().setDfSession(getDfSession());

 String strQuery = "SELECT name FROM dm_format WHERE can_index =
 true and topic_format = '0000000000000000' ORDER BY name";
```



```
// set the query on the list control
dropDownList.getDataProvider().setQuery(strQuery);
...
}
```

## Creating a validator

You can build a custom validator in the following steps:

1. [Developing a validator tag, page 633](#)
2. [Developing a validator class, page 634](#)
3. [Using the validator in a component, page 634](#)

The following example illustrates the steps in building a simple password validator that rejects passwords 1111 or 1234.

## Developing a validator tag

First, add the JSP to your custom tag library. In the following example, you have added your custom tag library definition file (\*.tld) to /WEB-INF/tlds, and you add an entry for the password validator:

```
<tlibversion>1.0</tlibversion>
<jspversion>1.1</jspversion>
<shortname>acme</shortname>
<tag>
 <name>passwordvalidator</name>
 <tagclass>com.mycompany.PasswordValidatorTag</tagclass>
 <bodycontent>jsp</bodycontent>
 <attribute>
 <name>name</name>
 <required>true</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
 <attribute>
 <name>controltovalidate</name>
 <required>true</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
 <attribute>
 <name>nlsid</name>
 <required>false</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
 <attribute>
 <name>errormessage</name>
```

```
 <required>false</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
 <attribute>
 <name>visible</name>
 <required>false</required>
 <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>
```

Create the tag class that extends `BaseValidatorTag`. Add accessor methods for any custom validator attributes:

```
package com.mycompany;
import com.documentum.web.form.control.validator.BaseValidatorTag;

public class PasswordValidatorTag extends BaseValidatorTag
{
 protected Class getControlClass()
 {
 return PasswordValidator.class;
 }
}
```

## Developing a validator class

Your validator control class must override `doValidate()` to provide custom validation:

```
package com.mycompany;
import com.documentum.web.form.control.validator.BaseValueValidator;

public class PasswordValidator extends BaseValueValidator
{
 protected boolean doValidateValue(String strValue)
 {
 boolean bValid = true;
 if (strValue.equals("1111") || strValue.equals("1234"))
 {
 bValid = false;
 }
 return bValid;
 }
}
```

## Using the validator in a component

Add the new passwordvalidator tag to the change password form:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ page errorPage="/wdk/errorhandler.jsp" %>
<%@ taglib uri="/WEB-INF/tlds/dmform_1_0.tld" prefix="dmf" %>
```

```

<%@ taglib uri="/WEB-INF/tlds/acme_1_0.tld" prefix="dmf" %>
<html>
<head>
<dmf:webform/>
<title><dmf:label label="ChangePassword"/></title>
</head>

<body class='contentBackground' topmargin='10' bottommargin='10'
 leftmargin='13' rightmargin='0' marginheight='10' marginwidth='13'
 onload='relogin()'>

<dmf:form>
<h3><dmf:label label="Change Password"/></h3>
<table>
 <tr>
 <td>Username:</td>
 <td><dmf:text name="userName" size="40"/>
 <dmf:requiredfieldvalidator name="val"
 controltovalidate="userName" errorMessage=
 "You must supply a name"/></td>
 </tr>
 <tr>
 <td>Current Password:</td>
 <td><dmf:text name="password" size="40"/>
 <dmf:requiredfieldvalidator name="val" controltovalidate=
 "password" errorMessage= "You must supply your current
 password"/></td>
 </tr>
 <tr>
 <td>New Password:</td>
 <td><dmf:text name="newpassword" size="40"/>
 <dmf:requiredfieldvalidator name="val" controltovalidate=
 "newpassword" errorMessage= "You must enter a new password"/>
 <acme:passwordvalidator name="val" controltovalidate=
 "newpassword" errorMessage="You cannot use 1111 or 1234"/>
 </td>
 </tr>
</table>
</dmf:form>
</body>
</html>

```

## Creating a qualifier

Custom qualifier classes must implement the `IQualifier` class and the following methods:

**`getContextNames()`** — Returns a string array of context names for the scope.

**`getScopeName()`** — Returns the name of the scope associated with the qualifier

**getScopeValue(QualifierContext)** — Returns the scope value for one or more context values. For example, for the scope "type", the caller passes in an object ID of "0193c...3e7d", and the method returns "dm\_document".

**getParentScopeValue(String)** — For a passed scope value, returns the parent scope value. For example, for the scope "type", the caller passes in a scope of 'dm\_document', and the method returns 'dm\_sysobject'. A null return value indicates that there is no parent.

**getAliasScopeValues(String)** — Returns an array of equivalent alias values for a passed scope value

## Using a prompt within a container

If your component needs to use a prompt to display a warning or confirmation, nest to the prompt component. Your component must be a container class in order to use the prompt component, that is, it must extend Container or a subclass of Container. The Container class implements a listener interface so that you can call setComponentNested().

### To display a prompt in a container

1. Call the prompt component in your component event handler. For example, if you component has an OK button, add the following event handler.

```
public void onOkPressed(Form form, Map map)
{
 ArgumentList args = new ArgumentList();
 //Get the prompt window title from your component NLS resource
 args.add(Prompt.ARG_TITLE, getString("MSG_CONFIRM_COMMIT_TITLE"));
 //Get the prompt message from your component NLS resource
 args.add(Prompt.ARG_MESSAGE, getString("MSG_CONFIRM_COMMIT_MESSAGE"));
 //Add the path to the warning icon
 args.add(Prompt.ARG_ICON, Prompt.ICON_WARNING);
 //Add button text
 args.add(Prompt.ARG_BUTTON, new String[] {Prompt.CONTINUE,
 Prompt.CANCEL});
 setComponentNested("prompt", args, getContext(), this);
}
```

2. Check prompt state in your component class when the prompt component returns.

```
public void onReturn(Form form, Map map)
{
 //check the Don't show again checkbox
 String strDontShow = (String)map.get(Prompt.RTN_DONTSHOWAGAIN);
 if (strDontShow !=null &&
 Boolean.valueOf (strDontShow).booleanValue())
 {
 //Call a helper method that inhibits the prompt next time
 inhibitPrompt();
 }
}
```

```
 removeReturnValue(Prompt.RTN_BUTTON);
 removeReturnValue(Prompt.RTN_DONTSHOWAGAIN);
 }
 //check which button was clicked
 String strButton = (String)map.get(Prompt.RTN_BUTTON);
 if (strButton != null && strButton.equals(Prompt.CONTINUE)
 {
 ...
 setComponentReturn();
 }
}
```

3. Inhibit the prompt if the Don't Show Again checkbox was checked:

```
private void inhibitPrompt()
{
 IPreferenceStore preferences = PreferenceService.
 getPreferenceStore();
 preferences.writeBoolean(INHIBIT_PROMPT_PREFERENCE,
 Boolean.TRUE);
}
```



508, *see* accessibility

## A

Accelerated Content Services

configuration, 66

accessibility

applets, 589

buttons and icons, 585

configuration in app.xml, 65

control label, 586

development guidelines, 584

event handlers, 586

frames, 589

images, 587

login, 585

service, 584

tables, 588

writing alt tags and descriptions, 590

<accessibility>, 65

ACS, 66

actionlist, 178

actions

action execution with pre- and  
post-processing, 629

configuration file, 215

controls, 389

custom, 625

dynamic state, 172

execution, 493

filtering clipboard functions, 599

generic, 172, 215

genericnoselect, 173

implementation, 627

introduction, 211

LaunchComponent, 496

LaunchComponent navigation, 239

launching, 212

launching components, 239

listeners, 500

lists, 178

multiselect, 172

nesting, 503

overview, 211

passing arguments, 213

preconditions, 491

role-based, 292

roles and object ACL permissions, 293

service, 491

singleselect, 172

tracing, 337, 628

version, 55

<adobe\_comment\_connector>, 72

alttextenabled, 310

app.xml, 58

AppFolderName, 58

applet, 91

applets

accessibility, 589

installation fails, 330

application

failover, 89

qualifier, 485

application connector

authentication, 271

components, 266

events, 270

application layer

contents, 39

inheritance, 42

overview, 38

application stops working, 328

<application>, 60

applications

configuration example, 309

configuration file, 58

configuration in app.xml, 60

environment properties, 88

environments, 43

inheritance, 42

interaction of elements, 46

login, 103

- managing frames, 120
- name, 58
- packaging and deployment, 154
- performance, 344
- timeout, 101
- tracing, 339
- architecture, Documentum stack, 26
- ArgumentTag, 433
- ArrayResultSet, 405
- asynchronous
  - actions, 572
  - components, 574
  - framework, 575
  - global settings, 82
  - listeners, 575
  - overview, 571
  - process, 577
  - UI, 577
- attribute lists
  - in data dictionary, 199
- attributelist
  - configuration file, 196
  - context or scope, 195
  - controls, 192
- attributes
  - configuration example, 312
  - custom data handler, 407
  - custom data handlers, 77
  - display of, 395
  - single and repeating, 193
  - tracing, 338
- authentication, 539
  - elements, in app.xml, 63
  - J2EE, 104
  - schemes, 540
- <authentication>, 63

## B

- binding
  - data, 186
  - to a thread, 550
- BOF, in WDK, 609
- branding
  - adding themes to NLS, 125
  - configuration example, 307
  - images and icons, 133
  - overview, 122
  - service, 578
  - style sheets, 127

- tracing, 337
- breadcrumb
  - displaying, 412
  - supporting, 413
- browser
  - history, 241
  - requirements, 70
- <browserrequirements>, 70
- browsertree, limiting size, 88
- business objects, in WDK, 609
- buttons, 134
  - accessible, 585
  - changing text, 134
  - configuring, 295

## C

- cache size
  - query, 346
- cachesize, 57
- caching data, 394
- classpath, 323
- client capability
  - overview, 287
  - roles, 289
- client session state
  - global setting, 78
- <client-sessionstate>, 78
- ClientCacheControl, 85
- clientenv, 53
  - qualifier, 485
- clipboard
  - filtering actions, 599
  - overview, 595
  - tracing, 337
  - using in a component, 597
- clusters
  - failover, 89
- columns
  - configuring, 229
  - dynamic (runtime), 232
- columns, number of, 185
- combocontainer, 246
- comment
  - Adobe, settings in app.xml, 72
- compiling
  - JSP pages, 157
- component
  - getting reference in JSP, 623
  - guidelines, 359



- implementing failover, 442
  - included, 449
  - listener, 447
  - preferences, 273
- componentinclude, 240
- components
  - adding parameters, 615
  - APIs, 435
  - calling components from URL, 247
  - calling from action, 248
  - Component class, 436
  - containers, 243
  - creating a definition, 615
  - creating JSP pages, 616
  - customizing, 614
  - definition, 221
  - dispatcher, 467
  - dispatching, 467
  - extending, 614
  - filters, 228
  - hiding with notdefined, 227
  - hiding with scope, 227
  - implementation, 445
  - included, 240
  - inheritance, 224
  - invoking within a container, 247
  - jump to, 437
  - launch by action, 239
  - lifecycle, 469
  - messages and labels, 236
  - navigation methods, 437
  - navigation overview, 237
  - overview, 219
  - reuse, 436
  - role-based UI, 294
  - scope, 225
  - sequence of processing, 470
  - tracing, 339
  - UI, 233
  - updating with client events, 421
  - URL bridge, 469
  - URL to, 238
  - using static HTML or JSP, 237
  - using the clipboard, 597
  - version, 55
  - WDK 5 bridge, 468
- compression, content transfer, 66
- CompressionFilter, 85
- <config>, 60
- ConfigResultSet, 404
- ConfigService, 480
- configuration
  - adding menu items, 300
  - application startup, 309
  - attributes example, 312
  - branding example, 307
  - buttons and images, 295
  - classes, 479
  - component XML file, 221
  - control tags, 160
  - data grid, 302
  - defined, 33
  - element, in app.xml, 60
  - examples, 295
  - files, 49
  - files, finding, 162
  - inheritance, 51
  - lookup algorithm, 489
  - object filter example, 317
  - overview, 47
  - processing, 488
  - properties example, 311
  - scope (qualifiers), 52
  - service, 479
  - tag libraries, 164
  - tracing, 337
  - validator example, 308
- configuration file
  - action, 215
  - application, 58
  - attributelist, 196
  - component, 221
  - lookup, 481
- containers
  - abstract, 245
  - accessing contained components, 462
  - calling components in a container, 247
  - calling contained components from
    - script, 247
  - calling from server class, 457
  - change notifications, 458
  - classes, 457
  - combo, 246
  - components that require, 250
  - content transfer, 247
  - dialog, 245
  - labels, 249
  - modal, 251
  - navigate to next page, 441
  - navigating, 460

- navigation, 245
- overview, 243
- propagating values within, 460
- property sheet, 246
- tracing, 339
- wizard, 245
- content transfer
  - applet, 91
  - containers, 247
  - debugging, 535
  - fails, no temp directory, 330
  - listeners, 531
  - modes, compared, 510
  - overview, 509
  - progress, 537
  - registry, 527
  - results (UCF), 531
  - service and processor classes, 533
  - stream to browser, 537
  - using 5.2.5 components, 536
- content transfer service
  - classes, 532
- ContentTransferService, 533
- <contentxfer>.<common>, 66
- context
  - attributelist, 195
  - introduction, 31
  - overview, 487
  - passing values to action, 215
  - relation to scope, 52
- contextvalue attribute, 215
- Control class
  - base class, 380
  - class properties, 380
- control tag
  - configuration, 160
  - tracing, 337
- controls
  - action-enabled, 171, 389
  - adding events, 624
  - attribute lists, 192
  - boolean, 170
  - changing a style, 130
  - choosing a superclass, 624
  - classes, 379
  - component helper, 170
  - configuration overview, 160
  - Control base class, 380
  - ControlTag class, 381
  - creating, 383
  - customizing, 623
  - databound, 184, 389
  - databound, see databound
    - controls, 186
  - Documentum object validation, 204
  - event arguments, 168
  - events, 165
  - format, 170
  - handling events on client, 168
  - handling events on server, 420
  - hiding, 177
  - ID, 384
  - indexed, 385
  - lifecycle, 428
  - media, 170
  - multiple selection, 415
  - naming, 384
  - passing arguments to event
    - handler, 433
  - relation to tags, 432
  - retrieving values, 384
  - scrollable, 183
  - setting values, 386
  - string input, 170
  - tooltips, 208
  - tracing, 337
  - types, 169
  - using, 382
  - validation, 202
  - XML configuration file, 175
- ControlTag, 381
- CookieManager, 567
- cookies
  - overview, 99
  - writing and reading, 567
- copy\_operation
  - in app.xml, 79
- <copy\_operation>, 79
- CreateInstallerWAR tool, 155
- cross-site scripting
  - HTML in attribute values, 201
  - validation, 71
- <custom\_attribute\_data\_handlers>, 77
- customization
  - components, 614
  - controls, 623
  - defined, 33
  - object grid, 617
  - validator, 633

**D**

- data binding
  - support classes, 390
- data dictionary
  - attribute labels, 137
  - list of attributes, 192
  - refresh, 355
  - scopes, displayed in WDK, 195
  - search, 144
  - using attribute lists, 199
- databound controls
  - binding data, 186
  - caching data, 394
  - data grid configuration example, 302
  - data handler classes, 406
  - DataProvider class, 390
  - getting data, 390
  - introduction, 184
  - overview, 389
  - paging, 189
  - result sets, 404
  - ResultSet data provider, 390
  - sorting, 188
- datadropdown lists
  - configuring, 181
- datagrid
  - adding custom attributes, 407
  - compared to objectgrid, 613
- DataProvider
  - getXXX, 393
  - setQuery, 391
- DataProvider class, 390
- dates, future, 327
- debugging
  - content transfer, 535
  - Java, 357
  - JavaScript, 356
  - JSP, 355
  - overview, 354
  - search, 558
  - XML, 356
- <default-mechanism>, 66
- defaultonenter event, 167
- deployment, 156
- deployment descriptor
  - listeners, 85
  - servlets, 86
- DFC
  - storing objects, 549
- DFC business objects, 329
- directory structure
  - applications, 38
  - branding, 124
- <discussion>, 77
- display
  - global settings, 81
- <display>, 81
- dmcl
  - tracing, 334
- docaseattributelist
  - lookup, 403
- docbase
  - qualifier, 484
- DocbaseAttributeCache, 550
- docbaseattributelist control, 194
- docbaseobject
  - configuration, 395
- DocbaseObjectCache, 550
- docbaseobjectconfiguration
  - definition, 395
- <docbaseobjectconfiguration>, 395
- Documentum type
  - custom icons, 296
- drag and drop
  - global setting, 79
  - in component definition, 452
  - in control, 455
  - in JSP, 454
  - overview, 450
  - performance, 454
  - supported components, 451
  - troubleshooting, 456
- <dragdrop>, 79, 452
- dragdropregion tag, 454
- dropdown lists
  - configuring, 181
- dynamicAction.js, 172

**E**

- entitlement
  - qualifier, 486
- Environment.properties, 88
- environments
  - for development, 37
  - introduction, 43
- error message
  - configuration in app.xml, 70
  - service, 601

- `<errormessageservice>`, 70
  - event
    - Content Server, notification, 97
  - event handler
    - accessibility, 586
    - navigation, 113
    - navigation to component, 112
    - registering client handlers, 114
    - setting programmatically, 427
  - events
    - arguments, 168
    - between frames, 117
    - client preprocessing, 242
    - client to server, 417
    - client-side, overview, 112
    - control, 165
    - control lifecycle, 428
    - control state change, 423
    - control, configuring, 168
    - handle on server or client, 417
    - handling inter-frame events, 119
    - handling inter-frame events on server, 119
    - handling on client, 168
    - handling on server, 420
    - how they are raised, 424
    - server to client, 422
    - updating components, 421
  - execution
    - class, 493
    - element, 217
    - permit, 217
  - `<execution>`, 217
  - ExpandInstallerWAR tool, 156
- ## F
- failover, 89
    - enabling for the application, 89
    - implement in component, 442
    - tracing, 336
  - `<failover>`, 62
  - `<fallback_identity>`, 62
  - filters, 228
    - based on role, 293
    - dynamic (with `LaunchComponent`), 498
    - latency/bandwidth, 350
    - servlet, 85
  - flags, tracing, 335
  - foreign object, 464
  - foreign objects
    - operations on, 242
  - foreign type, 218
  - form processor
    - navigation operations, 473
    - overview, 470
    - properties, 97
  - `<formats>`, 74
  - FormInclude, 475
  - FormOperation, 473
  - forms
    - class, 475
    - custom, 475
    - form processor, 470
    - tag class, 475
    - WebformIncludes s (generate style sheets and JavaScript), 475
  - FormTag, 475
  - fragment control, 189
  - `<fragmentbundles>`, 189
  - frames
    - accessibility, 589
    - event handlers, 119
    - firing events, 117
    - handling events on server, 119
    - managing, 120
  - framework
    - WDK, 571
- ## G
- generic, 172
  - generic actions, 215
  - genericnoselect, 173
  - global registry, 486
- ## H
- help service
    - calling, 592
    - overview, 590
  - history
    - browser, 241
    - configuring, 98
    - performance, 349
    - setting size, 241
  - hooks
    - form navigation, 97
    - lookup, 482

HTML  
 in attributes, 201  
 HttpContentTransportManager, 529

## I

IApplicationListener, 550  
 IAuthenticationScheme, 540  
 IConfigElement, 479  
 IConfigLookup, 481  
 IConfigLookupHook, 482  
 IConfigReader, 483  
 icons  
 controls, 207  
 for custom type, 296  
 themes, 133  
 IDfSession, 545  
 IDfSessionManager, 547  
 IDfSessionManagerEventListener, 551  
 ILaunchComponentEvaluator, 498  
 images, 207  
 buttons, 134  
 configuring, 295  
 image service, 579  
 in style sheets, 128  
 replacing, 207  
 themes, 133  
 included component, 449  
 <infomessageservice>, 70  
 inheritance  
 applications, 42  
 components, 224  
 configuration, 51  
 strings, 139  
 <init-controls>, 534  
 initialize  
 content transfer controls, 534  
 inlinecompression, 66  
 <inlinecompressionXXX>, 66  
 input mask  
 utility, 606  
 validator, 203  
 interaction  
 of application elements, 46  
 internationalization  
 locale service, 580  
 testing, 141  
 invalid ticket, 332  
 IParams, 476  
 IQualifier, 484

IRequestListener, 551  
 IReturnListener, 447  
 ISessionListener, 551

## J

J2EE authentication, 104  
 Java  
 debugging, 357  
 Java properties files, *see* properties files  
 java.io.IOException, 327  
 java.lang.verify, 332  
 JavaScript, 327  
 debugging, 356  
 files in WDK, 116  
 generated, 116  
 getting component value, 121  
 modal windows, 425  
 postServerEvent, 417  
 registering, 115  
 tracing, 117  
 using, 115  
 utilities, 112  
 <job-execution>, 82  
 JSP  
 comments, 156  
 creating, 236  
 debugging, 355  
 fragments, 189  
 getting component reference, 623  
 implicit objects, 553  
 jump to page, 437  
 request tracing, 336  
 return URL, 473  
 standard, 25  
 structure of page, 233  
 using outside components, 257  
 jump  
 to component, 437  
 to component page, 437

## K

keyboardnavigationenabled, 310

## L

language  
 element, in app.xml, 62 to 63  
 <language>, 62 to 63  
 latency

- performance filters, 350
- LaunchComponent, 215, 239, 248, 496
  - dynamic filters, 498
  - with permit check, 218
- lifecycle
  - component, 469
  - control events, 428
- listener
  - component, 447
- listeners
  - action, 500
  - application, 550
  - content transfer, 531
  - control, 430
  - in web.xml, 85
  - request, 550
  - session, 550
  - session manager, 551
- <listeners>, 78
- ListResultSet, 405
- lists, dropdown
  - configuring, 181
- load balancing, 352
- locale
  - element, in app.xml, 62
- <locale>, 62
- locales
  - adding, 138
  - adding localized files, 140
  - login, 111
  - NLS file naming, 140
  - NLS files, 138
  - retrieving strings, 141, 581
  - service, 580
  - service APIs, 583
  - tracing, 337
- logging, 324, 343
  - UCF, 523
- login, 103
  - accessible, 585
  - explicit, 111
  - J2EE principal, 104
  - locale, 111
  - manual, 104
  - password encryption, 105
  - preferences, 111
  - setting up J2EE principals, 106
  - silent, 542
  - skip authentication, 110
  - ticketed, 108

## M

- max\_sessions, 112
- memory allocation, 346
- menus
  - configuration example, 300
  - configuring, 178
  - passing arguments to, 180
  - role-based, 507
- message service, 600
- mirror object, 464
- modal windows
  - overview, 425
  - performance, 353
- <modified\_vdm\_nodes>, 76
- move\_operation
  - in app.xml, 80
- <move\_operation>, 80
- multiple selection
  - actions, 172
  - implementing, 415

## N

- navigation, 329
  - component methods, 437
  - components, 237
  - history, 241
  - in client event handler, 113
  - LaunchComponent, 239
  - return to caller, 440
  - to a component, 112
  - to next page, 441
  - URL parameters, 424
  - URL to component, 238
  - within containers, 460
- NLS
  - adding new themes, 125
  - definition, 137
  - dynamic substitution, 582
  - file names, 140
  - includes, 138
  - nlsbundle element, 583
  - strings, 138
- noReturnURL, 473
- notdefined, 227
- notification
  - Content Server event, 97
- <notification>, 72

**O**

- objectfilter
  - example, 317
- objectgrid
  - compared to datagrid, 613
  - custom, 617
- olecompound element, 217
- onchange event, 167
- onclick event, 167
- onselect event, 167
- out (JSP object), 554

**P**

- page not found
  - HTTP 1.1 not enabled, 328
- PageContext, 553
- paging
  - of data, 189
- password encryption, 105
- performance
  - actions, 345
  - browser history, 349
  - cookies, 349
  - drag and drop, 454
  - events, 344
  - HTTP sessions, 348
  - import, 352
  - latency/bandwidth filters, 350
  - memory settings, 346
  - modal windows, 353
  - object creation, 345
  - overview, 344
  - paging and cache size, 346
  - preferences, 349
  - qualifiers, 352
  - queries, 345
  - strings, 346
  - tracing, 345
  - value assistance, 349
- permit
  - precondition, 218
- plugins
  - in app.xml, 80
- <plugins>, 80
- postComponentJumpEvent(), 113
- postComponentNestEvent(), 114
- postServerEvent(), 417
- precompiling, 157
- precondition
  - element, 216
  - overview, 491
  - passing argument to, 389
  - permission level, 218
  - role, 216
- <precondition>, 216
- preferences
  - component, 273
  - component, storing and retrieving, 566
  - configuration, 276
  - custom component, 563
  - definition, 273
  - login, 111
  - overview, 563
  - storing and retrieving, 567
  - tracing, 337
  - user, configuring, 277
- <preferences>, 273
- <preferred\_renditions>, 75
- primary element, 50
- primary folder path
  - displaying, 412
  - getting, 412
- principal (J2EE) authentication, 104
- privilege
  - qualifier, 485
- progress
  - content transfer, 537
- properties
  - configuration example, 311
  - not updated with data dictionary change, 325
- properties files
  - accessibility support, 587
  - adding to application, 140
  - configuring, 138
  - dynamic substitution in, 582
  - inheritance, 139
  - naming, 140
  - overriding strings, 141
  - overview, 56
  - retrieving strings, 581
  - string externalization, 137
- propertysheetcontainer, 246
- proxy
  - reverse, configuration for, 522
- <pseudo\_attributes>, 201
- pseudoattributes, 201

**Q**

qualifiers  
 clientenv, 53  
 element in app.xml, 60  
 performance, 352  
 resolving to scope, 484  
 <qualifiers>, 60  
 queryExecute(), 492

**R**

reference object, 464  
 refresh  
 configuration and data dictionary, 355  
 data, 394  
 value assistance, 205  
 XML, 49  
 registerClientEventHandler(), 114  
 registry  
 in content transfer, 527  
 repeating attributes  
 editing, 193  
 replica object, 464  
 request (JSP object), 553  
 RequestAdapter, 85  
 <requestvalidation>, 71  
 response (JSP object), 554  
 result sets, 404  
 ArrayResultSet, 405  
 ConfigResultSet, 404  
 ListResultSet, 405  
 ScrollableResultSet, 404  
 TableResultSet, 405  
 return navigation, 440  
 return values  
 from a container, 448  
 rich text  
 configuration, 190  
 richtexteditor  
 in app.xml, 80  
 <richtexteditor>, 80  
 role  
 qualifier, 485  
 <rolemodel>, 64  
 roles  
 actions based on, 292  
 client capability plugin, 289  
 component filter, 293  
 configuration, 287  
 constraints, 293

in Webtop, 287  
 menus for, 507  
 model in app.xml, 64  
 role model adaptor, 506  
 role plugin, 291  
 RoleService class, 505  
 service, 505  
 tracing, 337  
 UI based on, 294  
 runatclient attribute, 167

**S**

SafeHTMLString, 604  
 scope  
 and qualifiers, 52, 484  
 attributelist, 195  
 components, 225  
 element in app.xml, 60  
 foreign, 218  
 hiding components, 227  
 hiding subtypes, 218  
 <scope>, 60  
 scrollable controls  
 configuring, 183  
 ScrollableResultSet, 404  
 search  
 advanced, configuration, 147  
 basic, configuration, 146  
 configuring, 144  
 customization, 557  
 debugging, 558  
 preferences, configuring, 153  
 results, configuring, 151  
 using 5.2.5 search, 154  
 search controls  
 configuring, 145  
 servlet  
 filters, 85  
 servlets  
 component dispatcher, 467  
 in web.xml, 86  
 mapping in web.xml, 83  
 standard, 26  
 WDKController, 442  
 session management  
 configuration in app.xml, 73  
 <session\_config>, 73  
 SessionManagerHttpBinding, 547  
 sessions



- cookies, 352
  - HTTP and repository, 545
  - IdfSessionManager, 547
  - repository, 545
  - SessionManagerHttpBinding, 547
  - state, 549
  - tracing, 552
  - sessions, running out of, 329
  - setComponentJump(), 457
  - setComponentNested(), 457
  - setComponentReturn, 440
  - setReturnError, 602
  - shortcutnavigationenabled, 310
  - Show All Properties, 325
  - singleselect, 172
  - skip authentication, 110
  - SSL
    - configuring UCF support, 518
  - startupAction, 212
  - StaticPageExcludes, 89
  - StaticPageIncludes, 89
  - storing objects, 549
  - strings
    - configuration overview, 137
    - dynamic substitution in, 582
    - inheritance, 139
    - retrieving, 141, 581
  - StringUtil, 605
  - style sheets
    - changing a control style, 130
    - identifying, 130
    - images, 128
    - introduction, 127
    - modifying, 130
    - order of precedence, 127
    - webforms.css, 129
  - <supported\_locales>, 62
- T**
- TableResultSet, 405
  - tables
    - accessibility, 588
  - tabs
    - configuring, 181
  - tags
    - attributes, 382
    - base classes, 381
    - generating UI, 409
    - relation to controls, 432
    - using libraries, 164
    - WebformIncludes (generate style sheets and JavaScript), 475
  - testing
    - internationalization, 141
  - testing components, 324
  - themes
    - adding to NLS, 125
    - configuration in app.xml, 64
    - definition, 123
    - directory structure, 124
    - images and icons, 133
    - overview, 122
    - processing, 126
  - <themes>, 64
  - thread
    - binding and caching, 550
    - tracing, 336
  - ThreadLocalCache, 550
  - ThreadLocalVariable, 550
  - timeout
    - control, 102
    - J2EE server setting, 101
    - overriding, 102
  - Tomcat
    - slows down, 328
  - tooltips, 208
  - tracing, 324, 628
    - adding flags, 342
    - asynchronous jobs, 340
    - client-side, 342
    - clipboard, 337
    - components and applications, 339
    - content transfer, 341
    - controls, 337
    - dfc, 334
    - dmcl, 334
    - framework operations, 337
    - JavaScript, 117
    - JSP processing, 338
    - overview, 333
    - performance, 345
    - sessions, 336, 552
    - turning on, 333
    - UCF, 341
    - virtual links, 339
    - WDK flags, 335
  - type
    - qualifier, 484

**U**

- ucf
  - client configuration, 515
- UCF
  - client config files, 514
  - client path substitution, 517
  - client services, 513
  - customization, 533
  - logging, 523
  - process, 525
  - results, 531
  - server config files, 521
  - server services, 520
  - troubleshooting, 524
- <ucfrequired>, 520
- UcfSessionInit, 85
- URL
  - in JSP pages, 235
  - parameters, 476
  - to component, 238
  - to component in container, 238
  - virtual link encoding, 96
- user.home, 515
- UseVirtualLinkErrorPage, 96

**V**

- validation, 202
  - base control class, 429
  - by form processor, 428
  - configuration example, 308
  - custom control, 633
  - input mask, 203
  - of objects, 429
  - process, 428
  - tracing, 338
- value assistance
  - non-data dictionary, 205
  - overview, 204
  - performance, 349
  - refresh, 205
- version
  - components and actions, 55

- qualifier, 485
- view
  - content in browser, 537
- virtual links
  - error handling, 96
- virtual document
  - tracing the tree, 339
- virtual links
  - authentication, 93
  - deployment, 92
  - overview, 91
  - path resolution, 94
  - URL encoding, 96

**W**

- WAR files
  - introduction, 45
  - packaging tool, 155
- WDKController, 85, 442
- web.xml
  - filters, 85
- WebformIncludes, 475
- WebformTag, 475
- WebLogic
  - compiler failure, 326
  - content transfer fails, 332
  - java.io.IOException, 327

**X**

- <xforms>, 77
- XML
  - configuration files, 49
  - debugging, 356
  - file checkin, 332
  - file extensions, configuring, 73
  - file import, 331
- <xmlfile\_extensions>, 73
- XSS, *see* cross-site scripting

**Z**

- ZipArchive, 606