

EMC® Documentum® Web Development Kit

Version 6.5

**Development Guide
P/N 300-007-231 A01**

EMC Corporation
Corporate Headquarters:
Hopkinton, MA 01748-9103
1-508-435-1000
www.EMC.com

Copyright © 2001 - 2008 EMC Corporation. All rights reserved.

Published July 2008

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED AS IS. EMC CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.

For the most up-to-date listing of EMC product names, see EMC Corporation Trademarks on EMC.com.

All other trademarks used herein are the property of their respective owners.

Table of Contents

Preface	19
Chapter 1 Configuring WDK Applications	23
Introduction	23
WDK Architecture	24
The WDK architectural stack	25
Content Server	27
Java EE application server	27
Service layer.....	27
The WDK environment layer.....	27
Presentation model.....	28
Component model.....	28
Application model.....	29
Client.....	30
Finding files to configure or customize.....	30
Parts of a WDK configuration file	32
Modifying configuration elements	33
Modifying definitions: syntax	33
Inserting and replacing elements	35
Differences between modifying and extending definitions.....	37
Modifying and extends: how they work together.....	37
Modifying elements by ID.....	38
Extending XML definitions	40
Scoping a feature	41
Scopes available in WDK applications.....	42
Scoping features for a specific context	44
Scoping features for a specific client environment.....	45
Scope, qualifier, and Context classes	46
Chapter 2 Administering an Application	47
Configuring content transfer options	47
Setting content transfer and ACS/BOCS options	48
Scanning Microsoft formats for linked documents	51
Configuring UCF.....	52
Configuring UCF on the client.....	52
Installing executables with UCF.....	57
Configuring UCF client path substitution.....	58
Configuring UCF support for unsigned or non-trusted SSL certificates	59
Windows client registry in UCF.....	61
Configuring UCF on the application server	62
Configuring UCF support for proxy servers	63
Enabling login and session options	63
Configuring authentication options	64
Supporting saved credentials	65
Setting trusted domains to prevent redirects.....	65

Specifying supported languages	66
Configuring Java EE principal authentication	67
Encrypting passwords	69
Configuring theme availability	70
Enabling accessibility (Section 508)	70
Supporting roles and client capability	71
Using Content Server roles	71
Using client capability roles.....	72
Using both Server roles and client capability	74
Configuring high availability support	74
Configuring timeout and number of sessions	74
Enabling and configuring application-wide features	76
Enabling retention of folder structure and objects on export	76
Enabling modal pop-up windows.....	77
Configuring email import	77
Enabling the EMCMF format in WDK-based applications	78
<messageArchive-support> configuration elements	79
Log messages	79
Enabling shortcuts (hotkeys)	80
Enabling datagrid features	81
Enabling autocompletion	81
Configuring presets access	82
Configuring a preference repository and preference cookies.....	83
Enabling and disabling session state	84
Enabling redirect security	84
Enabling the streamline view	85
Enabling notifications	85
Enabling application failover support	86
Configuring application-wide failover	86
Configuring component failover.....	87
Configuring anonymous virtual link access.....	87
Virtual links overview.....	89
Enabling discussion sharing	90
Configuring hidden objects and invalid actions display.....	90
Supporting asynchronous jobs.....	91
Configuring applet display	92
Configuring the copy operation.....	93
Configuring the move operation.....	93
Configuring navigation defaults	94
Setting web-link type in Application Connectors	94
Configuring browser history	95
Registering custom classes	96
Configuring lightweight sysobject parent display	97
Enabling and configuring drag and drop and spell check	97
Configuring the Active-X plugin install.....	97
Configuring drag and drop	98
Enabling the rich text spell checker	99
Configuring file format, rendition, and PDF Annotation Services support.....	100
Mapping extensions to formats.....	100
Supporting XML file extensions.....	101
Specifying preferred renditions and their viewing and editing applications.....	101
Enabling PDF Annotation Services	103
Configuring the application environment.....	104
Configuring application properties	104
Specifying supported browsers.....	105

Turning on cross-site scripting security	106
Configuring the client environment	107
Configuring type icon display	109
Configuring web deployment descriptor settings.....	109
Naming the application	110
Specifying static pages for better performance	110
Reducing HTTP sessions.....	111
Specifying error pages	111
WDK servlet filters	111
WDK servlets	112
WDK listeners	114
Application layers	114
Contents of an application layer	116
Chapter 3 Configuring and Customizing Controls	119
Control configuration overview.....	119
Types of controls	120
Action-enabled controls	121
How value assistance is rendered	123
Configuring controls.....	124
Finding the file for configuring a control	124
Configuring a control.....	126
Global control configuration.....	127
Configuring tabs.....	129
Configuring tab key order	130
Configuring action visibility and behavior	131
Validating user input	133
Configuring autocompletion	134
Configuring context (right-click) menus	134
Creating fixed menus.....	135
Filtering by object type.....	139
Configuring rich text	140
Configuring scrollable panes	142
Configuring JSP fragments	143
Adding a tooltip	144
Adding images and icons	145
Hiding a control	146
Control customization overview.....	146
Control classes	146
How controls and tags work together	149
Types of control events	150
Choosing server-side or client-side event handling	152
Choosing a control superclass	153
Customizing controls.....	153
Getting controls programmatically	153
Accessing contained controls.....	155
Accessing controls by JavaScript	155
Validating a control value.....	156
Creating a validator control.....	157
Implementing multiple selection	159
Customizing drop-down lists	160
Adding autocompletion support to a control	161
Modifying the display and handling of attributes	162
Configuring a required attribute.....	162
Creating an attribute formatter	163
Creating an attribute value handler.....	163

	Creating a custom attribute tag class	164
	Adding a custom element to be used in attribute handling	166
	Adding an attribute editor	166
	Parts of the docbaseobjectconfiguration file	167
	Default attribute handling	168
	Adding a control listener.....	169
	Generating control UI	170
	Getting and displaying the folder path.....	171
	Programming control events	172
	Control events	172
	Handling a control event in the component class	173
	Firing a server event from the client.....	174
	Firing a client event from the server	176
	Managing frames.....	176
	Handling a client-side control event.....	177
	Handling navigation from a client-side function	178
	Registering a client event handler	179
	Using client-side scripts	179
	Firing events between frames	181
	Making a control accessible to JavaScript.....	183
Chapter 4	Configuring and Customizing Data Access	185
	Databound controls	186
	Configuring data display and selection	186
	Configuring column headers and resizing	186
	Configuring datagrid row selection	188
	Customizing row selection	190
	Configuring data sorting.....	192
	Configuring datagrid flow and folder refresh	193
	Configuring data paging.....	193
	Configuring drop-down lists	194
	Configuring attribute display	195
	Generating lists of attributes for display.....	196
	Configuring the attributelist control.....	196
	Creating a context-based attribute list	197
	Creating an attributelist that is independent of the data dictionary	198
	Configuring a "Starts with" filter.....	201
	Providing data to databound controls	202
	Accessing datagrid controls.....	203
	Getting data in a component class.....	204
	Getting data from a query	205
	Getting or overriding data in a JSP page.....	206
	Refreshing data	207
	Caching data	207
	Rendering data with result sets.....	208
	Adding custom attributes to a datagrid.....	210
	Supporting lightweight sysobject display in a component	212
	Implementing non-data dictionary value assistance	213
Chapter 5	Configuring and Customizing Actions	215
	Actions overview.....	215

Configuring actions	216
How to launch an action	216
Adding action controls to a JSP page.....	217
Passing arguments to actions.....	218
Configuring an action definition.....	219
Removing an action from a component	221
Hiding or displaying an action	222
Configuring and customizing shortcuts to actions (hotkeys)	222
Specifying a shortcuts mapping file	223
Creating a shortcut definition.....	223
Creating a shortcuts map	224
Adding shortcut support to a control	226
Customizing actions	227
Passing arguments to an action.....	228
Passing arguments to menu action items	229
Using the LaunchComponent execution classes	231
Providing action NLS strings.....	232
Creating a custom action definition.....	232
Using dynamic filters in action definitions	233
Creating an action precondition.....	236
Commonly used action preconditions	238
Implementing action execution.....	239
Adding an action listener	241
Nesting actions.....	245
Pre- and post-processing actions.....	245
Chapter 6 Configuring and Customizing Components	249
Configuring components	249
Modifying or extending a component definition	249
Versioning a component.....	250
Scoping a component definition	251
Hiding a component for specific contexts	252
Hiding features within a component.....	253
Calling a component by URL.....	254
Calling a component from an action.....	255
Calling a component from JavaScript	256
Calling a component from another component	257
Calling a component that launches a startup action.....	258
Including a component in another component.....	258
Configuring data columns in a component	259
Adding or removing static data columns.....	260
Configuring dynamic data columns.....	262
Creating a component JSP page	264
Using JSP pages outside a component.....	264
Configuring messages and labels.....	265
Operating on a foreign object	265
Processing a form before submission	265
Configuring locators.....	266
Adding custom help for a component	267
Configuring Application Connector components	269
Modifying the Documentum menu.....	269
Removing menu items	269
Modifying menu items.....	270
Adding custom menu items	272
Restricting menu items to specific applications	272
Application connector components and actions	274
Adding Application Connector components and actions	274
The appintgcontroller component.....	275

Managing events	277
Managing authentication	278
Customizing components.....	280
Navigating within a component	280
Jumping or nesting to another component.....	281
Returning to the calling component	282
Returning to a component, then jumping to another	283
Passing arguments from a control to a component	283
Passing arguments between components.....	284
Accessing an included component	285
Implementing a component.....	285
Rendering messages to users	287
Reporting errors	289
Supporting export to CSV	291
Implementing failover support	292
Adding a component listener	295
Supporting clipboard operations	297
Supporting drag and drop.....	298
Adding drag and drop to a JSP page.....	299
Adding drag and drop support to a control.....	301
Troubleshooting drag and drop	301
Getting a component reference in a JSP page	302
Using modal windows	303
Configuring and customizing containers.....	304
Calling a container by URL, JavaScript, or action	304
Requiring a component visit within a container	306
Calling a container from a server class	306
Implementing container notifications.....	307
Accessing components within containers	309
Passing arguments in a container.....	311
Navigating within a container	312
Using combocontainer	313
Using a prompt (pop-up) within a container.....	313
Creating modal containers	315
Containers overview.....	316
Choosing a container	317
Using Business Objects in WDK.....	318
Components overview	319
Component features	319
Component definition (XML)	321
Component JSP pages.....	323
Component tag classes.....	323
Component processing	324
Chapter 7 Configuring and Customizing Multi-Component Features	329
Configuring and customizing preferences	329
Preferences and cookies	329
Configuring user preferences	330
Configuring user column display preferences.....	331
Customizing user preferences	334
Bypassing user preferences	337
Configuring a component-level preference	337
Using custom presets.....	343
About presets.....	343
Using presets in a component.....	344
Looking up preset data within a component.....	345

Troubleshooting presets	346
Configuring and customizing strings	346
Adding locales	347
Adding strings to properties files.....	347
Inheriting strings	347
Naming properties files.....	348
Overriding strings in the UI	349
Displaying escaped HTML strings	349
Designing for and testing internationalization	350
Retrieving localized strings	354
Adding dynamic messages in NLS strings.....	355
Branding an application	356
Creating a new theme	356
Registering a custom theme.....	357
Using style sheets	358
Modifying a style sheet	359
Adding images and icons	360
Configuring buttons	362
How themes are located by the branding service	363
Creating asynchronous jobs.....	364
Asynchronous action job execution.....	364
Asynchronous component job execution	366
Job execution framework	367
UI in asynchronous processing	368
Asynchronous process	369
Customizing authentication and sessions	369
The authentication service.....	370
Getting a session in a component or action class	371
Getting a session in any WDK class.....	372
Storing and retrieving objects in the session	374
Clearing DFC and WDK object caches.....	374
Binding and caching in a request thread.....	375
Adding an application, session, or request listener.....	375
Listening to DFC sessions.....	376
Synchronizing a session	377
Supporting multi-repository sessions.....	377
Managing a multi-repository session	378
Adding multi-repository support to a component.....	379
Scoping and preconditioning actions on remote objects.....	380
Using replica (mirror), reference, and foreign objects	380
Tracing sessions.....	381
Adding an authentication scheme	381
Supporting silent login.....	383
Chapter 8 Configuring and Customizing Search	387
Understanding search in WDK.....	387
Configuring search controls	389
Configuring basic search.....	390
Configuring advanced search	391
Configuring search results and tuning performance.....	394
Making custom attributes available in search results	395
Customizing search in Webtop applications	396
Modifying the search JSP pages.....	397
Modifying the search component query	401
Hiding the customization from query editing	404

	Programmatic search value assistance.....	404
Chapter 9	Configuring and Customizing Content Transfer	407
	Content transfer modes compared	407
	HTTP content transfer.....	410
	Streaming content to the browser	412
	UCF overview	412
	Initializing content transfer controls.....	412
	Configuring UCF components.....	413
	Customizing a UCF component.....	414
	Troubleshooting UCF.....	415
	Logging UCF.....	416
	Getting return values from content transfer	418
	Content transfer service classes.....	419
	Displaying content transfer progress.....	420
	Logging and tracing content transfer component operations.....	420
Chapter 10	Configuring and Customizing Roles and Client Capability	423
	Role-based actions	423
	Setting role precedence	424
	Role-based filters	425
	Role-based UI.....	426
	Custom role plugin.....	426
	Role configuration overview	427
Chapter 11	Customizing the Application Framework	429
	Application elements interaction	429
	Configuration service overview	430
	Configuration service classes.....	430
	ConfigService.....	431
	IConfigContext.....	431
	Configuration lookup	432
	Configuration lookup hooks.....	433
	Configuration reader	434
	Context.....	435
	Configuration service processing.....	436
	Lookup algorithm.....	437
	Improving performance	437
	Action implementation	438
	Documentum object creation	439
	String management.....	439
	Paging	439
	Java EE memory allocation.....	440
	HTTP sessions.....	441
	Preferences.....	442
	Browser history	442
	Value assistance.....	442
	Search query performance.....	443
	High latency and low bandwidth connections	443
	Qualifiers and performance.....	445

Import performance.....	445
Load balancing.....	445
Modal windows and performance	446
Tracing	446
Turning on WDK tracing.....	446
Using DFC tracing	447
Adding custom tracing flags	447
Using client-side tracing.....	447
Logging	448
Testing components.....	449
Utilities	449
Using the comment stripper utility	452
Making an application accessible	453
Making labels accessible.....	453
Providing image descriptions	453
Adding frame titles.....	454
Accessibility mode.....	454
Troubleshooting runtime errors	455
Error loading main component.....	456
Show All Properties does not work	456
Properties do not display after data dictionary change	456
JavaScript error on application connection.....	457
"Configuration base has not been established"	457
Application no longer starts after code change.....	457
Application slows down	457
Page not found errors in if HTTP 1.1 not enabled in client browser	458
Application runs out of sessions	458
Browser navigation renders actions or links invalid	458
Cannot import an XML file.....	458
Unable to locate checked out objects after deploying a WDK-based application	459
Controls don't display any repository data	459
Tags all have the same label.....	459
Custom authentication.....	460
Using ticketed login.....	460
Skipping authentication for a component	462
Chapter 12 Configuring and Customizing Webtop	465
Webtop directory structure.....	465
Webtop views.....	465
Using Webtop qualifiers.....	467
Interaction of inbox and task manager components.....	468
Creating bookmark URLs (favorites).....	471
Redirecting to Webtop	472
Customizing the Webtop browser tree	472
Using Webtop navigation utilities	476
Appendix A WDK Tracing Flags	479
Tracing sessions.....	479
Tracing WDK framework operations.....	480
Tracing controls and validation.....	481

Tracing JSP processing	482
Tracing components and applications	484
Tracing virtual links.....	485
Tracing servlets	486
Tracing asynchronous operations.....	486
Tracing content transfer	487
WDK trace flags to trace Test Harness at runtime.....	488
Appendix B Configuration Settings in WDK-based Application Deployment	489

List of Figures

Figure 1.	WDK physical layout.....	25
Figure 2.	WDK-based application architectural stack	26
Figure 3.	UCF sample client configuration mapping	53
Figure 4.	Application layers and configuration inheritance.....	115
Figure 5.	Properties tab bar	129
Figure 6.	Custom tab bar.....	129
Figure 7.	Home cabinet with standard files filter.....	140
Figure 8.	Home cabinet with custom filter	140
Figure 9.	Scrollable pane controls	143
Figure 10.	Control and ControlTag relationship.....	149
Figure 11.	Client-side and server-side event processing	151
Figure 12.	String attribute rendered as text control	164
Figure 13.	String attribute rendered as TextArea control	166
Figure 14.	Scoped configuration.....	252
Figure 15.	Serialization process	292
Figure 16.	Pop-up prompt.....	315
Figure 17.	Component processing sequence	325
Figure 18.	JSP page, control, and user interaction	326
Figure 19.	WDK display preferences	332
Figure 20.	Column selector component.....	333
Figure 21.	Custom type column preferences.....	334
Figure 22.	Debug preferences.....	351
Figure 23.	Custom theme.....	358
Figure 24.	Custom type icon	361
Figure 25.	Job execution interaction diagram.....	369
Figure 26.	Authentication service interfaces	371
Figure 27.	Authentication scheme processing.....	382
Figure 28.	Search size custom drop-down list.....	390
Figure 29.	Limiting the selectable types and subtypes.....	392
Figure 30.	Limiting the selectable types without subtypes.....	392
Figure 31.	Conditional value assistance UI.....	393
Figure 32.	Attribute selection drop-down.....	398
Figure 33.	Specific attributes as search criteria.....	398
Figure 34.	Custom attributes as search criteria.....	399
Figure 35.	Full-text search box removed from UI.....	401
Figure 36.	Simple search without added clause	402
Figure 37.	Simple search with added clause	402

Figure 38. Content transfer component classes and service layer 420

Figure 39. Main view in Webtop..... 466

Figure 40. Inbox processing sequence 468

Figure 41. Viewing a task in the task manager 469

Figure 42. Finishing a task 470

Figure 43. Browser tree **Add Repository** option 474

Figure 44. Add Repository option is hidden..... 476

Figure 45. **Repositories** tab removed 476

List of Tables

Table 1.	Presentation model services	28
Table 2.	Modification elements.....	34
Table 3.	Differences between modification and extension.....	37
Table 4.	Content transfer default mechanism, ACS, and BOCS settings (<code><contentxfer></code> element).....	48
Table 5.	Proxy settings.....	51
Table 6.	UCF client configuration settings.....	53
Table 7.	Settings in <code>ucf.client.config.xml</code>	57
Table 8.	Content transfer registry keys used by Windows registry mode UCF content transfer	61
Table 9.	UCF application server configuration settings	62
Table 10.	Authentication elements (<code><authentication></code>).....	64
Table 11.	Save credentials elements (<code><save_credentials></code>)	65
Table 12.	Trusted domains element (<code><trusted-domains></code>)	65
Table 13.	Language elements (<code><language></code>)	66
Table 14.	Theme elements (<code><themes></code>)	70
Table 15.	Accessibility elements (<code><accessibility></code>).....	71
Table 16.	Client capability mapping to <code>dm_user</code>	72
Table 17.	Client capability roles	72
Table 18.	Session management elements (<code><session_config></code>).....	75
Table 19.	Modified VDM action timeout (<code><modified_vdm_nodes></code>).....	76
Table 20.	Modal window elements in <code>app.xml</code> (<code><modalpopup></code>)	77
Table 21.	Email conversion settings (<code><messageArchive-support></code>)	79
Table 22.	Autocomplete elements in <code>app.xml</code> (<code><auto_complete></code>).....	81
Table 23.	Presets elements (<code><presets></code>)	82
Table 24.	Preferences configuration elements.....	83
Table 25.	Event notification elements (<code><notification></code>)	85
Table 26.	Documentum Collaborative Services elements (<code><discussion></code>).....	90
Table 27.	Asynchronous job elements (<code><job-execution></code>).....	91
Table 28.	Applet tag elements (<code><applet-tag></code>)	92
Table 29.	Copy operation elements	93
Table 30.	Move operation elements	94
Table 31.	Navigation settings.....	94
Table 32.	Listener elements (<code><listeners></code>).....	96
Table 33.	Process Builder Forms Builder elements (<code><xforms></code>)	96
Table 34.	Displaying lightweight sysobjects (<code><lightweight-sysobject></code>)	97
Table 35.	Active-X plugins elements (<code><plugins></code>)	97

Table 36.	Components that support drag and drop	98
Table 37.	Drag and drop elements (<dragdrop>)	99
Table 38.	Rich text editor elements (<richtexteditor>).....	99
Table 39.	Formats elements (<formats>).....	100
Table 40.	XML extensions elements (<xmlfile_extensions>).....	101
Table 41.	Preferred renditions elements.....	102
Table 42.	PDF Annotation Services elements (<adobe_comment_connector>).....	103
Table 43.	Environment settings.....	104
Table 44.	Browser requirement elements (<browserrequirements>)	105
Table 45.	URL request validation elements (<requestvalidation>).....	106
Table 46.	Client environment elements (<clientenv> and <clientenv_structure>).....	108
Table 47.	Server environment elements (<serverenv>)	108
Table 48.	Client session state elements (<client-sessionstate>).....	109
Table 49.	Static page context parameters.....	110
Table 50.	<errorpage>.....	111
Table 51.	WDK filters.....	112
Table 52.	WDK servlets	112
Table 53.	Deployment descriptor listener.....	114
Table 54.	Main directories and files in a WDK-based application	116
Table 55.	State of a control based on dynamic attribute value.....	123
Table 56.	Control global configuration	128
Table 57.	Control visibility based on context.....	131
Table 58.	Menu configuration elements	136
Table 59.	Rich text configuration elements.....	140
Table 60.	Rich text editor configuration elements (<editor>).....	141
Table 61.	Control class properties	147
Table 62.	Base tag classes.....	147
Table 63.	Event attributes	150
Table 64.	Choosing a control superclass	153
Table 65.	Default attribute handling.....	168
Table 66.	Folder path display rules	172
Table 67.	WDK scripts.....	180
Table 68.	Row selection settings.....	189
Table 69.	Sample Documentum Application Builder scope definitions	198
Table 70.	Hotkeys configuration elements	224
Table 71.	Keys that can be used in a shortcut combination	225
Table 72.	Binding scenarios for versioned components	250
Table 73.	Data column configuration elements.....	259
Table 74.	Application Connectors menu configuration elements	271
Table 75.	Appintgcontroller <dispatchitems> elements.....	276
Table 76.	Required pages in appintgcontroller component definition	276
Table 77.	Application Connector control events	277
Table 78.	Configuration elements (<dragdrop>	298

Table 79.	WDK drop target actions.....	300
Table 80.	Conditions for selecting WDK containers	317
Table 81.	User preference components	330
Table 82.	Column display preference elements	331
Table 83.	<preference> elements.....	337
Table 84.	Feature support in content transfer modes	408
Table 85.	Client configuration settings in content transfer modes	409
Table 86.	Server configuration settings in content transfer modes.....	409
Table 87.	UCF component definition elements	413
Table 88.	UCF content transfer result variables	418
Table 89.	Uses of role in configuration.....	428
Table 90.	Comment stripper utility parameters	452
Table 91.	Components and pages that are loaded by the main Webtop component.....	466
Table 92.	Session tracing flags.....	479
Table 93.	Framework tracing flags	480
Table 94.	Control and validation tracing flags	481
Table 95.	Form and page tracing flags	483
Table 96.	Component and application tracing flags	484
Table 97.	Virtual link tracing flags.....	485
Table 98.	Servlet tracing flags	486
Table 99.	Asynchronous operation tracing flags	486
Table 100.	Content transfer tracing flags	487
Table 101.	Mandatory configuration before deployment	489
Table 102.	Optional configuration before deployment.....	490
Table 103.	Optional configuration after deployment	490

Preface

This development guide for the Web Development Kit (WDK) and client applications is task-oriented and focuses on the user's task, not the user's role. This guide is designed to help you with three types of tasks: administration, configuration, and customization.

Administration — Administration tasks include the following:

- Installation, which is described in the *Web Development Kit and Webtop Deployment Guide*
- Management of users, groups, security, audits, jobs, management of formats and types, and indexing, which are performed using Documentum Administrator and described in *Content Server Administration Guide*
- Configuring application-wide settings, which are documented in this guide, in [Chapter 2, Administering an Application](#).

Prerequisites for administration configuration tasks

You must be familiar with editing of XML files and, for certain tasks, the application server environment.

Configuration — Configuration is defined for support purposes as changing an XML file or modifying a JSP page to configure controls on the page. Configuration does not require a developer license.

Configuration is described in this developer guide. With WDK 6, some tasks that previously required customization in 5.3, such as creating a query for display in a component, are externalized to a configuration task in WDK 6.

A separate reference guide (WDK Reference Guide) describes all configurable features of WDK controls, actions, and components.

Prerequisites for general configuration tasks

You must be familiar with the following languages and standards:

- JavaServer Pages and Servlet technologies, including tag libraries, in the versions that are 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

Customization — Customization is defined for support purposes as extending WDK classes or modifying the JSP pages to include new functionality. Customization requires a developer license.

WDK-related Java development and general customization tasks that apply to more than one component are described in this guide. A separate reference guide, WDK Reference Guide, describes some customizations of specific WDK components.

Prerequisites for customization tasks

You must be familiar with the Java language, in the version that is specified in the release notes for your product.

Documentation resources

WDK contains documentation and source files to assist you in developing custom web applications.

- This guide
 - Contains general configuration, customization, and application-building information for application developers.
- *Web Development Kit and Webtop Reference*
 - Contains information about all of the configurable settings for controls, actions, and components in WDK, with some additional component-specific customization notes.
- *Web Development Kit Tutorial*
 - This tutorial contains several modules on setting up a development environment and configuring and customizing WDK.
- *Web Development Kit and Webtop Deployment Guide*
 - This guide describes how to prepare for, modify, 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 located in `wdk/src`.
 - Source files for all webcomponent layer components and actions are located in `webcomponent/src`. For Webtop deployments, source files for Webtop components and actions are located in `webtop/src`.
- Javadoc API references
 - This documentation is available on the product download site and includes the Javadoc API reference sets for the WDK, webcomponent, and Webtop Java libraries.

- The EMC Developer Network website, developer.emc.com

This website 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 website.

Check the EMC Documentum technical support website (powerlink.emc.com) for revisions of the documentation. After you log in to Powerlink, click the menu **Support Knowledgebase Search Documentation and White Papers Search** to search for documents related to your version of WDK or WDK client application. The support site also provides peer support forums that are monitored by technical support experts.

Revision history

The following changes have been made to this document.

Revision Date	Description
July 2008	Initial publication for 6.5.

Configuring WDK Applications

This chapter introduces you to the design of applications based on Web Development Kit (WDK) and general configuration tasks.

The topic [WDK Architecture, page 24](#) describes the features of a WDK-based application.

This chapter addresses the following common configuration tasks. There is no prescribed sequence to the tasks, but typical custom applications must apply all of these tasks at one time or another.

- [Finding files to configure or customize, page 30](#)
- [Parts of a WDK configuration file, page 32](#)
- [Extending XML definitions, page 40](#)
- [Modifying configuration elements, page 33](#)
- [Scoping a feature, page 41](#)
- [Scoping features for a specific client environment, page 45](#)

Introduction

The Web Development Kit (WDK) Automated Test Framework helps automate the generation of and running reusable test cases for WDK applications. You can use the WDK Automated Test Framework to perform these kinds of tests on WDK applications:

- Sequence of user actions
- Validate actions
- Validate the states of controls

To ensure reusable test cases, the WDK Automated Test Framework does not depend on nor test the locations of user interface elements on the browser page.

WDK Architecture

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.

The following topics describe WDK architecture in more detail:

- [WDK foundation technologies, page 24](#)
- [The WDK architectural stack, page 25](#)

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.

- Java EE Servlet Technology

A web application runs in a Java EE-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.

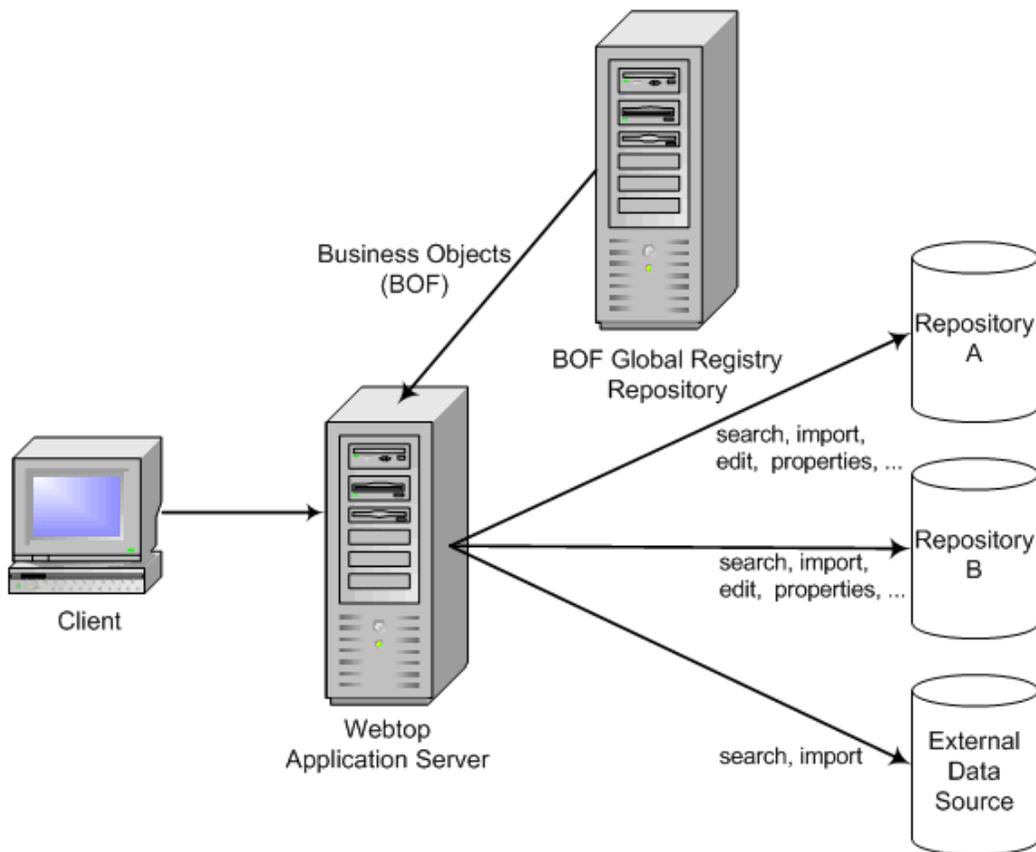
- Java EE security

If you set up Java EE security in your Java EE server, 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

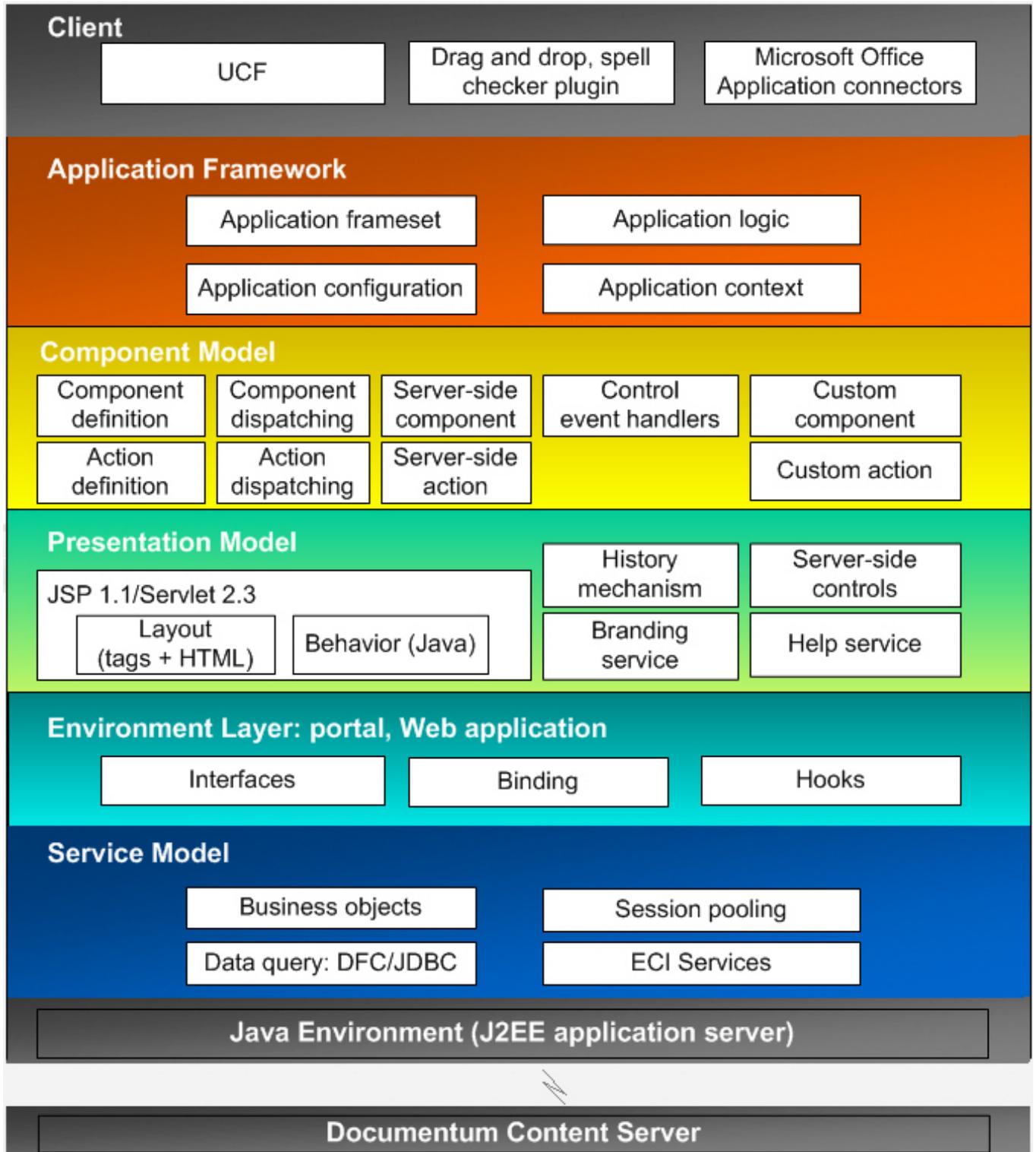
The physical components of a Documentum stack are shown in [Figure 1, page 25](#):

Figure 1. WDK physical layout



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

Figure 2. WDK-based application architectural stack



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.

Java EE application server

The Java EE application server provides Java EE 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 and DFC 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 `dfc.properties` file on the WDK host can be configured as a client for server connection pooling (`dfc.session.pool.enable`, was `connect_pooling_enabled` setting in 5.x `dmcl.ini`). WDK applications take advantage of session pooling, increasing performance over non-pooled sessions.
- Other services: ECI, Collaboration, Retention Policy, Media Transformation, Content Intelligence.

The WDK environment layer

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

- Stand-alone web application, such as Webtop, Documentum Administrator, Digital Asset Manager, or Web Publisher
- Portal environments (refer to release notes for certified versions)

Presentation model

The presentation model consists of JSP tag libraries and JavaScript 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 services show in [Table 1, page 28](#).

Table 1. Presentation model services

Service	Description
Form processor	Interprets HTTP requests and translates them into WDK method calls and events
History mechanism	Maintains browser history and navigation
Configuration service	Looks up configuration contracts for actions and components and dispatches the appropriate UI for the user's context and presets
Branding service	Manages the look and feel for the application
Locale service	Looks up localized UI text
Help service	Delivers localized, context-sensitive help
Message service	Displays localized messages to the user
Drag and drop support	Configurable support for drag and drop between repository locations and between desktop and repository

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.

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 WDK 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.

WDK 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.

- Application Connectors

WDK provides connectors that enabled Documentum functionality within specific Windows applications, such as Microsoft Word. For information on customization of Application Connectors, refer to *Application Connectors Software Development Kit Guide*.

The Java EE-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. This supports access to repositories and Webtop functionality from within the application

Finding files to configure or customize

Controls are configured in a JSP page. Actions and components, as well as some cross-component features, are configured in an XML configuration file.

- 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. Configure controls through the JSP tag attributes on the JSP page itself and, for certain controls, through XML files. (Refer to [Global control configuration, page 127](#) for details on XML control configuration.)

- JSP pages (forms)

Forms are JSP pages that contain a `<dmf:webform>` or `<dmf:form>` tag. Component JSP pages must contain one of these tags, and the top-level page must contain the `<dmf:webform>` tag. Generally there is a one-to-one correspondence between a form and a web page. You configure forms by changing the form layout in the JSP page itself. For more information on component JSP pages, refer to [Creating a component JSP page, page 264](#).

- Actions

A WDK action associates a UI event such as menu selection with an operation. 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 XML file containing the action definition and an action class that implements the action and determines whether a user can perform the 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. Configure actions in the action configuration file and configure a control to launch the action. For more information on configuring and customizing actions, refer to [Chapter 5, Configuring and Customizing Actions](#).

- 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 component definition, a behavior class, an NLS resource file, a help file, and at least one JSP page (the layout page). Configure components by changing the component definition or any other of the resources in the component definition. For more information on configuring and customizing components, refer to [Chapter 6, Configuring and Customizing Components](#).

- 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. For more information on using events to control application behavior, refer to [Programming control events, page 172](#).

- 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 inherit their configuration from a parent application layer. For example, the webcomponent layer inherits its application definition from the wdk layer. Configure your application in the 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. For more information on application layers, refer to [Application layers, page 114](#).

- Branding

The branding service manages the UI look by themes, which incorporate images and icons, and cascading style sheets (CSS). Configure branding by creating or modifying a theme, and register your theme in the application configuration file app.xml. Users select a theme for display in the Preferences component. For more information on branding, refer to [Branding an application, page 356](#).

- 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 customizing strings, page 346](#)). 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.

Java EE 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.

Parts of a WDK configuration file

The following elements are generally present in all WDK configuration files:

```

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

```

1 `<?xml>`: Standard XML language declaration.

2 `<config>`: The root element of the configuration file. The entire configuration definition is contained within this element. The version attribute is reserved for future changes to the WDK configuration framework.

3 `<scope>`: 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 [Scoping a feature, page 41](#) for more information on scope.

4 `<primary-element>`: Element that represents a definition, such as `<action>`, `<component>`, `<application>`, `<attributelist>`, or `<docbaseobjectconfig>`. You can specify more than one primary element within a `<scope>` element. Each type of primary element, such as application or component, has additional elements that are required by the particular kind of primary element. The elements common to all primary-element definitions are described in the following definitions.

5 `modifies`: Modifies an existing definition by adding (with or without specifying the position), replacing, or removing a specific element of the specified definition. For more information, refer to [Modifying configuration elements, page 33](#). Use in preference to `extends`.

6 `extends`: Specifies a definition that is inherited by the current definition. Refer to [Extending XML definitions, page 40](#) for more information. For simple changes to configuration files, replace the `extends` argument with a `modifies` argument. For more information on modification, refer to [Modifying configuration elements, page 33](#).

7 `notdefined`: Configurations can hide elements that are defined in the parent scope by using the `notdefined` attribute. Refer to [Hiding a component for specific contexts, page 252](#) 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>

```

8 `version`: 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 a component, page 250](#).

9 <nlsbundle>: 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. [Configuring and customizing strings, page 346](#) provides more information on string configuration.

10 <filter>: 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. [Hiding features within a component, page 253](#) provides more information.

11 <filtered_element>: Element to which the filter is applied, for example, <enableshowall>

Modifying configuration elements

You can modify configuration elements of a WDK, Webtop, client application, or custom configuration file by inserting, replacing, or removing elements. Modifications to configuration files are described in the following topics.

Modifying definitions: syntax

The syntax for the modification definition is the following. [] indicates optional arguments. | indicates an alternative argument. Place the XML modification file in custom/config. The file name is not important.

```
<config>
  <scope scope-name=scope-value>
    <primary-element modifies="[primary-element-id:]
      config-file | [primary-element-id:] scope-clause">
      <insert [path=element-path]>
        <element>
      </element>
      </insert>
      <insertafter path=element-path>
        <element>
      </element>
      </insertafter>
      <insertbefore path=element-path>
        <element>
      </element>
      </insertbefore>
      <replace path=element-path>
        <element>
      </element>
      </replace>
      <remove path=element-path/>
    </primary-element>
  </scope>
</config>
```

Key:

- *scope-name*: Can be unspecified or can specify one or more applicable scope attributes and values. For more information, refer to [Scoping a feature, page 41](#).
- *primary-element*: Specifies the top configuration element, for example, <application>, <action>, or <component>. The value of the modifies attribute is the configuration filepath and filename or, if there is more than one primary element in the file, the primary element ID. If there are two or more definitions in the file for the same primary element with differing scopes, the scope must be specified here as the value of *scope-clause*.
- *element-path*: Specifies the path to the element that is being modified. The path is relative to the primary element using dot notation. For example, the path <primary><element1><element2 attr='value1'></element2></element1><primary> has the value of element1.element2[attr=value1]. The path is not required for <insert> command if you are inserting an element under the primary element.

Table 2, page 34 describes the modifications that are supported.

Table 2. Modification elements

Operation element	Description
<primary-element modifies=...>	<p>Primary element, must be a child element of <scope>. The modifies attribute identifies the primary element and either the configuration file that contains the definition or the scope. The scope is required when you extend a configuration file that contains more than one scope element. The primary element ID must be left out when the referenced primary element has no ID. For example, modifying an application:</p> <pre data-bbox="852 1186 1339 1249"><application modifies="webtop/app.xml"></pre> <p>Modifying a component:</p> <pre data-bbox="852 1344 1323 1407"><component modifies="main:webtop/config/main_component.xml"></pre> <p>Modifying an attributelist definition:</p> <pre data-bbox="852 1501 1339 1585"><attributelist modifies="attributelist:application='webcomponent' type='dm_document'></pre>

Operation element	Description
<insert>	Inserts one or more elements at the end of the referenced element's children. If no path is defined, the specified elements are inserted at the end of the specified primary element's children.
<insertbefore>	Inserts one or more elements before the referenced element
<insertafter>	Inserts one or more elements after the referenced element
<replace>	Replaces the referenced element with the specified element(s)
<remove>	Hides the referenced element from the current scope

Note: You must provide a unique path with a unique attribute value in order to use <insertbefore> or <insertafter>. Alternatively, use <insert>.

To remove a <filter> element, you must modify the element that contains the filter element by replacing the entire contents.



Caution: For modifications to app.xml, the modification file must be in the custom/config directory or one of its subdirectories. The file does not need to be named "app.xml," because the configuration service does not keep track of file names.

Inserting and replacing elements

Example 1-1. Inserting a new element in a component definition

This example modifies the Webtop browsertree component definition and inserts a new preferences node at the end of the nodes list. The browsertree definition in webtop/config has the following partial content:

```
<component id="browsertree" extends="foldertree:webcomponent/config/navigation/
  foldertree/foldertree_component.xml">
  <nodes>
    <docbasenodes>
      ...
      <node componentid='cabinets'>...</node>
      <entrynode>homecabinet_classic</entrynode>
    </docbasenodes>
  </nodes>
</component>
```

The modification definition inserts a node as follows:

```

<config version='1.0'>
<scope>
  <component modifies='browsertree:webtop/config/browsertree_component.xml'>
    <insert path='nodes.docbasenodes'>
      <node componentid='custom_node'>
        <icon>customicon_16.gif</icon>
        <label>Custom Thing</label>
      </node>
    </insert>
  </component>
</scope>
</config>

```

Example 1-2. Identifying the modification path by an attribute value

When you specify the path to the element that you wish to modify, you can use any attribute value that is unique for that element. For example, the following modification inserts a **Cancel checkout** menu item and removes the **Edit** menu item from the context menu (right-click menu) for `dm_sysobjects`, defined originally in `dm_sysobject_actions.xml`. A portion is shown here. Note that the `<filter>` element is not a primary element:

```

<menuconfig id="context-menu">
<filter clientenv='not appintg'>
  <actionmenuitem dynamic="multiselect" action="editfile" nlsid="MSG_EDIT" .../>
  <actionmenuitem dynamic="singleselect" action="view" nlsid="MSG_VIEW_FILE" .../>...

```

The modification references a path to the `<actionmenuitem>` element with the action attribute value of "editfile":

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<config version="1.0">
  <scope>
    <menuconfig modifies="context-menu:webcomponent/config/actions/dm_sysobject_actions.xml">
      <insertafter path="actionmenuitem[action=editfile]">
        <actionmenuitem dynamic="multiselect" action="cancelcheckout"
          nlsid="MSG_CANCEL_CHECKOUT" showifdisabled="true"
          showifinvalid="false"/>
      </insertafter>
    </menuconfig>
  </scope>
</config>

```

Example 1-3. Replacing an element in a component definition

The following example replaces the JSP start page in the Webtop advanced search component. This example modifies the Webtop configuration in `advsearchex_component.xml`, but note that the configuration does not specify a start page. The `<pages>` element is inherited from the parent definition in the file `advsearch_component.xml` in `webcomponent/config/library/search/searchex`:

```

<component ...>
...
  <pages>
    <start>/webcomponent/library/advancedsearch/advsearchex.jsp</start>
  </pages>
</component>

```

The modification syntax is as follows:

```
<component modifies="advsearch:webtop/config/advsearchex_component.xml">
  <replace path="pages.start">
    <start>/custom/advancedsearch/advsearchex.jsp</start>
  </replace>
</component>
```

Differences between modifying and extending definitions

Table 3, page 37 shows differences in configuration and results using the modification and extension configuration mechanisms.

Table 3. Differences between modification and extension

	Modification	Extension
Syntax	<component modifies="id:path_to_config"	<component id="id" extends="id:path_to_config"
Target	Modifies the definition that contains the element to be modified	Extends the top layer of a definition
Primary element	Modifies it	Creates a new one
Usage	Effects point changes	Inherits and overrides
Limitation	Path must yield a unique result (to a single element or to element with unique attribute)	Must extend the top layer

Modifying and extends: how they work together

When you modify a definition, you must modify the original definition in which the target element is defined, even if there are layers of extended definitions. Your modifications to the base layer will show up in your extended definition. For example, the login component definition in wdk/config defines a list of parameters. The webtop login component extends this definition but does not change the parameters list. If your customization adds a parameter, you can modify the WDK definition using <insert>, and the webtop login component will inherit this modification. The path attribute points to the actual element that you are modifying. This modification has the following syntax:

```
<component modifies="login:wdk/config/login_component.xml">
  <insert path=params>
    <param name="myparam" required="true"/>
  </insert>
</component>
```

```
</insert></component>
```

You can also modify a modification that was applied by another layer. For example, if the DCM application adds a new DCM menu to the webtop menubar using the modification mechanism, your application can insert its new items to this modified menu.

When the configuration service loads modifications and extensions, it loads them in lowest (wdk) to highest (webtop/custom) layers. If you have a custom2 on top of custom and they both modify the same element, then custom would get modified, then custom2 would modify that. After all the modifications are applied, the extensions are applied. For example, as soon as the configuration service finds an element from highest to lowest, it replaces the element of the lower extended definition with the element of the higher extended definition.

Example 1-4. Deleting an element from an extended definition

The following example shows the effect of deleting an element from a component configuration that has been extended. The hypothetical WDK component definition defines a component `<primary>` element with a child element `<test>` that contains two child elements:

```
<component id="component1">
  <test>
    <a>aaa</a>
    <b>bbb</b>
  </test>...</component>
```

The Webtop component extends this definition as follows:

```
<component id="component2" extends="component1:wdk/config/component1.xml">
  ...</component>
```

Your custom configuration removes one of the configuration elements from the base WDK definition, `<test>.<a>`. Note that you modify the actual configuration in which the element appears, not the definition in Webtop:

```
<component modifies="component1:wdk/config/component1.xml">
  <remove path="test.a"/></component>
```

This removes the element `<a>` from your component. If some other application extends the Webtop component2, it will not be affected by your modification.

Modifying elements by ID

You may need to modify an element that has several instances in an XML file, each instance differentiated by ID. The following example changes a radio control in the checkin component definition by changing the version to be the same on checkin by default. The other control initial values are unchanged. The original definition in the file `checkin_component.xml` located in `webcomponent/config/library/contenttransfer/checkin` is as follows:

```
<init-controls>
  <control name="minorversion" type="com.documentum.web.form.control.Radio">
    <init-property>
```

```

        <property-name>value</property-name>
        <property-value>true</property-value>
    </init-property>
</control>
<control name="newversion" type="com.documentum.web.form.control.Radio">
    <init-property>
        <property-name>value</property-name>
        <property-value>true</property-value>
    </init-property>
</control>
</init-controls>

```

The modification has the following syntax. Line breaks are added for display but are not in the original:

```

<component modifies="checkin:webcomponent/config/library/
contenttransfer/checkin/checkin_component.xml">
    <replace path="init-controls.control[
        name=newversion].init-property.property-value">
        <property-value>>false</property-value>
    </replace>
</component>

```

Because the control has only one property that is initialized, you can replace it. If it has two properties, they cannot be identified by ID. To change an element without ID with one or more sibling elements, replace the entire named parent element. For example, you want to modify a value in the following control configuration:

```

<control name="newversion" type="com.documentum.web.form.control.Radio">
    <init-property>
        <property-name>value</property-name>
        <property-value>true</property-value>
    </init-property>
    <init-property>
        <property-name>label</property-name>
        <property-value>New?</property-value>
    </init-property>
</control>

```

To change the value of the label init-property, you would do the following:

```

<component modifies="checkin:webcomponent/config/library/
contenttransfer/checkin/checkin_component.xml">
    <replace path="init-controls.control[
        name=newversion]">
        <init-property>
            <property-name>value</property-name>
            <property-value>>false</property-value>
        </init-property>
        <init-property>
            <property-name>label</property-name>
            <property-value>New?</property-value>
        </init-property>
    </replace>
</component>

```

Extending XML definitions

When you customize an existing WDK component or action, you must either modify or extend its configuration definition. Modifications are the recommended method because they will pick up changes in upgrades to WDK-based applications. For example, if a custom properties component extends the Webtop properties component and overrides the set of parameters, an upgrade to the WDK component with a new parameter could break the customization. If you modified the component by adding a parameter, you would get your parameter plus the new parameter from the upgrade. Refer to [Modifying configuration elements, page 33](#) for more information.

If you extend an XML 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. If you copy an element, its contents will be overridden by your element unless you use the modification mechanism. Refer to [Modifying configuration elements, page 33](#) for information about modification.

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_homecabinet" extends="webtop/config/  
homecabinet_classic_component.xml"
```

- The primary element value and filename 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">
```

- Filename or primary element name plus scope within the referenced definition. If the parent configuration file contains more than one scoped definition with the same ID, you must use this syntax. In the following example, a custom action extends a scoped action definition in the webcomponent application layer. The application attribute is required.

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

Note: If you extend or modify an XML definition in a configuration file that contains more than one definition, you must specify the scope in the extends or modifies clause.

Scoping a feature

Features can be scoped so they are presented only when the user's context or environment matches the scope definition. Scoped definitions are overridden by presets. For information on presets, refer to [Using custom presets, page 343](#).

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.

Using scope in a configuration file — Use the following rules to apply scope in a configuration file:

- 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 by an extends or modifies definition. For example, to require a different set of parameters for the child type, use a modifies attribute and add, remove, or replace parameters within the <params> element. If your definition extends another definition and has no <params> element, all parameters are inherited from the parent.

- 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 has the following combined scopes:

```
<scope type="dm_group" privilege="creategroup">
```

Note: If you are extending or modifying an XML definition in a configuration file that contains more than one scope, you must use specify the scope value in the extends or modifies attribute.

For example:

```
<action modifies="importrendition:application='webcomponent'  
  type='dm_sysobject'"/>
```

- 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. Use the modifies attribute in the app.xml definition, and place this definition in custom/config. For example:

```
<application modifies="webtop/app.xml:>  
<insert path="qualifiers">  
<qualifier>com.mycompany.MyQualifier</qualifier>  
</insert>  
</application>
```

Scopes available in WDK applications

WDK includes the following qualifiers in wdk/app.xml: repository (docbase), Documentum type, privilege, role, clientenv, application, entitlement, 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 remote object (not in the current repository) 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 [Scoping features for a specific client environment, page 45](#).
- **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="5.3">.
 - Set the container <bindingcomponentversion> to the same version as the scope on the component. For example, <bindingcomponentversion>5.3</bindingcomponentversion> on your custom export container means that the component will contain an export component whose scope is version="5.3".

- \EntitlementQualifier

Checks entitlement evaluation classes that are registered in Entitlement.properties which is located in WEB-INF/classes/com/documentum/web/formext/config. These classes must implement IEntitlementService and evaluate whether a given product is installed and enabled in the current repository. For example, the CollaborationEntitlementService class calls the Collaboration Service to check the global registry for installation of the Documentum Collaborative Services product files and provides the value "collaboration" for the entitlement scope. Refer to *Content Server Installation Guide* for information on global registries.

- ApplicationLocationQualifier

Matches context value "location" to the user's current navigation location in the repository.

Scoping features for a specific context

Webtop adds qualifiers for repository name and location in the browser tree. Other WDK-based applications add their own qualifiers. 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.

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.

- 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 has the following combined scopes:

```
<scope type="dm_group" 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.

Scoping features for a specific client environment

The `clientenv` qualifier allows you to scope a component based on the client environment such as browser, portlet, or Application Connector. 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 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 `checkin` component JSP page has a `clientenvpanel` control that is rendered only in the Application Connectors 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 `searchsources_preferences.jsp` page, the panel that allows the user to select search sources is hidden in the portal environment:

```
<dmfx:clientenvpanel environment='portal' reversevisible='true'>
  <div><dmf:radio name ='<%=SearchSourcesPreferences.>
```

```
CONTROL_PREF_FAVORITE_REPOSITORIES%>' group='
  searchLocationRef' onClick="onChangeSearchPref" /></div>
</dmfx:clientenvpanel>
```

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

Scope, qualifier, and Context classes

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.

Use the Context object (refer to [Context, page 435](#)) 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.

Administering an Application

This chapter addresses the common administration tasks for a WDK-based application.

- [Configuring content transfer options, page 47](#)
- [Configuring UCF, page 52](#)
- [Enabling login and session options, page 63](#)
- [Enabling and configuring application-wide features, page 76](#)
- [Enabling and configuring drag and drop and spell check, page 97](#)
- [Configuring file format, rendition, and PDF Annotation Services support, page 100](#)
- [Configuring the application environment, page 104](#)
- [Configuring web deployment descriptor settings, page 109](#)
- [Application layers, page 114](#)
- [Contents of an application layer, page 116](#)

Most of the administrative tasks described in this chapter are performed in your custom `app.xml` file, within the `<application>` element. For general instructions on how to customize this file, refer to [Extending XML definitions, page 40](#).

Configuring content transfer options

Content transfer in WDK-based applications supports two modes:

- HTTP content transfer

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.

- Unified Client Facilities (UCF), a lightweight client-based service

If the application is configured to use UCF content transfer, a UCF applet downloads to the client on the first call to a content transfer operation. The applet is lightweight so 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. The applet requests permission from the user to install the UCF runtime from the application server host.

The UCF mechanism is required to support Adobe Comment Connector.

For information on the features available with each mode, refer to [Content transfer modes compared, page 407](#). Because UCF provides many more features, it is the recommended mode.

Setting content transfer and ACS/BOCS options

For information on Accelerated Content Services (ACS) and Branch Office Caching Services (BOCS), refer to *Distributed Configuration Guide*. ACS and BOCS write options require UCF content transfer.

Enable ACS read and write in order to read and write content files directly to the Content Server, bypassing the application server. The metadata, but not the content, is read and written through the application server. In synchronous read or write operations, the session is locked until the transaction completes.

Note: The ACS write feature requires ACS write to be enabled in the global registry repository as well as in the target repository.

Note: ACS uses ticketed login. The creation time and expiration time of the ticket are recorded as UTC time. When a ticket is sent to a server other than the server that generated the ticket, the receiving server tolerates up to a three minute difference in time to allow for minor differences in machine clock time across host machines. System administrators must ensure that the machine clocks on host machines on which applications and repositories reside are set as closely as possible to the correct time.

If you enable asynchronous write operations, content is uploaded to a BOCS server when the user is located closer to a BOCS server than to the Content Server. The content is available to other users on the BOCS server before it is uploaded to the Content Server. However, the metadata may arrive at the Content Server before the content does, in which case a user attempting to view the content from the Content Server and not the BOCS server will receive an error message to try the operation later.

You can configure the content transfer mechanism and the following ACS and BOCS settings in `wdk/app.xml`, within the `<contentxfer>` element.

Table 4. Content transfer default mechanism, ACS, and BOCS settings (<contentxfer> element)

Element	Description
<code><default-mechanism></code>	Sets the default content transfer mechanism. Valid values: <code>ucf</code> <code>http</code>

Element	Description
<mechanism>	<p>You can specify different content transfer mechanisms for roles. In the following example, contributors use UCF and consumers use HTTP:</p> <pre data-bbox="789 407 1409 533"><filter role="contributor" > <mechanism>ucf</mechanism> </filter> <filter role="consumer" > <mechanism>http</ mechanism> </filter></pre>
<acs>	<p>Contains elements that configure Accelerated Content Services: <enable>, <attemptsurrogateget>, <maintainvirtuallinks>. Deprecated in D6, replaced by <acceleratedread>.</p>
<accelerated-read>	<p>Contains elements that configure Accelerated Content Services read operations.</p>
.<enabled>	<p>Set to true to enable Accelerated Content Services read operations in the application. Requires configuration of Accelerated Content Services on the network.</p>
.<attemptsurrogateget>	<p>Set to false to avoid connecting to ACS servers that require surrogate get, which fetches from a remote storage area. Set to true (default) to attempt to use all ACS servers.</p>
.<maintainvirtuallinks>	<p>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 affected by this flag, since they do not display content inline in the browser.</p>
<accelerated-write>	<p>Contains settings to enable and configure write operations to ACS servers and, if installed, BOCS servers</p>
.<enabled>	<p>Set to true to enable write operations to the ACS server. If false, and <accelerated-read>.<enabled> is true, only read operations will pass through the ACS server.</p>

Element	Description
.<bocs-write-mode>	<p>Enables or disables write operations through a BOCS server when the BOCS server is closer to the user than the ACS and Content Servers, and specifies the default mode. Values:</p> <ul style="list-style-type: none"> • prohibit-async (default) Turns off asynchronous write to BOCS server • default-syncDefault writes to BOCS and ACS servers simultaneously but allows asynchronous write when overridden by a user with override permission • default-asyncDefault writes asynchronously from BOCS server to ACS server when optimal but allows synchronous write when overridden by a user with override permission
<userresidentucfinvoker>	<p>Enables and disables:</p> <ul style="list-style-type: none"> • Automatically preloading the UCF client on client login, which enables better performance during content transfers • Having all UCF clients use a single javaw process, which enables more efficient use of system resources <p>The default is on (true).</p>
.<allow-override-bocs-write-mode>	<p>Allows scoped user group to override the default BOCS write mode. If this option is enabled, a UI option for immediate send or send through BOCS server is displayed in the UI. This element is filtered for one or more specified groups that have the override privilege.</p>

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

Table 5. Proxy settings

Element	Description
<clientinfo>	Sets global proxy client access.
<proxyclientipheader>	Specifies the name of the request header that an intermediate proxy server uses to pass the client IP address. Commonly used headers are x-forwarded-for (Apache), proxy-ip (Sun). Default = empty. If this field is empty, WDK will use request.getRemoteAddr() to obtain the client IP. When the request passes through multiple intermediate proxies, if this field contains multiple values, the first value is used as the client IP.

Scanning Microsoft formats for linked documents

For WDK-based applications that use UCF content transfer, scanning for linked documents in Microsoft compound documents can be enabled. Link scanning may have a performance impact, and it is not enabled by default for the application. The user must also set a preference to enable link scanning. This preference is not enabled by default.

When link scanning is enabled, the following formats will be scanned for links and the linked documents will be imported or checked in:

- Microsoft Word
- Microsoft Excel
- Microsoft Powerpoint

The following versions of these Office documents are supported:

- 97
- 2000
- XP
- 2003
- 2007

On import or checkin of a document with linked documents, a dialog will displayed asking if the user wants to scan for linked documents. When linked documents are imported, the parent and children are internally treated as virtual documents, but the user will not be able to reorder the linked documents within the compound document.

Note: Content linked with the Office hyperlinks feature is not included in an import or check-in.

To enable link scanning for the application

1. Open app.xml in the custom directory in a text editor or IDE.

2. Add the following lines:

```
<embedded-links-scan>  
  <enabled>true</enabled>  
</embedded-links-scan>
```

3. Save and close the file, and restart the application server.

The user must set the preference to enable link scanning, but this preference will not take effect unless link scanning is enabled for the application.

To disable link scanning for a specific content transfer component

1. Extend the action definition that launches a specific content transfer component, for example, `dm_sysobject_actions.xml`.
2. Copy the action definition into your extended action definition, for example, the checkout action.
3. Uncomment the following line:

```
<!--<olecompound enabled="false"/>-->
```

4. Save and close the file, and refresh the configurations in memory by navigating to `http:server/webtop/wdk/refresh.jsp`.

Configuring UCF

The following topics describe administrative tasks in UCF content transfer. For additional configuration and customization tasks that are performed by application developers, refer to [Chapter 9, Configuring and Customizing Content Transfer](#).

Configuring UCF on the client

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 if the Windows client option "registry.mode" in `ucf.installer.config.xml` is set to "windows" and not "file". Refer to [Windows client registry in UCF, page 61](#) 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 Webtop applications in the directory `wdk/contentXfer`. In DFC Web applications, you must place this file in any location on the application path that is directly accessible by clients, for example, `root_directory/contextXfer`.

The contents of the client configuration 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". A typical installation is diagrammed below.

Figure 3. UCF sample client configuration mapping

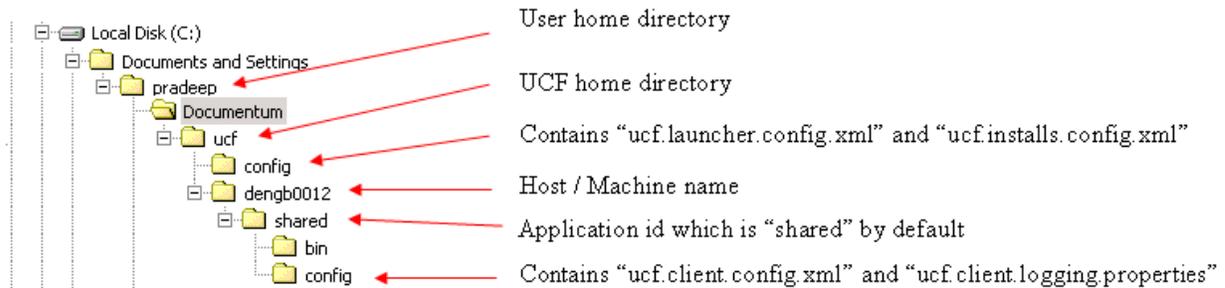


Table 6, page 53 describes the configurable settings in ucf.installer.config.xml located in WDK-based applications directory wdk/contentXfer. Non-configurable settings in the file are not listed in this table. Some settings listed here are not present in the default configuration file but can be added. You must change the version attribute value on the <app> element to force a refresh on the clients. For example, you can add a letter to the version: `version="6.5.0a"`.

Table 6. UCF client configuration settings

Element	Description
<platform>	Specifies platform details. The combination of os and arch values must be supported by the deployed 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 UCF 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 Configuring UCF client path substitution , page 58.

Element	Description
<ucfInstallsHome>	Location for components installed by UCF on the client. Refer to Configuring UCF client path substitution , page 58.
<executables>	Contains one or more <executable> elements that specify executables that will be installed by the UCF installer. For more information on installing executables, refer to Installing executables with UCF , page 57.
<configuration>	Contains <option> elements that configure default content transfer file locations on the client
<option name=*.dir>	Options whose name end in .dir specify a path. The path must be absolute or begin with a path substitution variable.
<option name=user.dir>	Sets base location for export.dir, checkout.dir, viewed.dir, temp.working.dir, and logs.dir
<option name=export.dir>	Default location for exported files (UI allows user to select location during export, saved as preference)
<option name=checkout.dir>	Location for checked out files
<option name=viewed.dir>	Location for viewed files
<option name=temp.working.dir>	Location for temporary upload download of files, cleaned up after user specifies the destination
<option name=logs.dir>	Location of client log file, used when client tracing is enabled in tracing.enabled option
<option name=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 a client file registry.xml. On all platforms except Macintosh, this file is located in \$user.dir/documentum. On Macintosh: \$user.dir/registry.xml
<option name=tracing.enabled>	Turns on client UCF tracing to a log file ucf.trace*.log where * is a number appended to the log name.

Element	Description
<option name= file.poll.interval>	Interval in seconds (non-negative) to poll for changes to UCF configuration files
<option name= 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 Configuring UCF support for unsigned or non-trusted SSL certificates, page 59 .
<option name= https.truststore.file> or https.keystore.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 Configuring UCF support for unsigned or non-trusted SSL certificates, page 59
<option name= https.truststore.password> or https.keystore.password	Specifies the encoded password for the trust store. Refer to Configuring UCF support for unsigned or non-trusted SSL certificates, page 59
<option name= https.truststore.password.encoding> or https.keystore.password.encoding	Specifies the encoding of the password for the trust store. Refer to Configuring UCF support for unsigned or non-trusted SSL certificates, page 59
<option name= https.truststore.encrypted.password>	Specifies the trust store password. Default=changeit. 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 Configuring UCF support for unsigned or non-trusted SSL certificates, page 59
<option name= cipher.name>	Name of cipher used to encrypt truststore password. Refer to Configuring UCF support for unsigned or non-trusted SSL certificates, page 59
<option name= cipher.secret.key>	Cipher key bytes encoded in base64. Refer to Configuring UCF support for unsigned or non-trusted SSL certificates, page 59
<option name= cipher.secret.key.algorithm>	Name of algorithm to use for cipher key. Refer to Configuring UCF support for unsigned or non-trusted SSL certificates, page 59
<option name= house.keeping.interval>	Add this setting to specify an interval in days after which viewed files on the client will be cleaned up. The housekeeping is performed at the next user login after the interval.

Element	Description
<option name=ucf.launcher.executablefile.pathcheck>	If set to true, checks for the existence of the viewing application in the system path. Default = true
<option name=optimal.chunk.size	Size of each chunk in chunked request/responses. Default of 48k works best for most application servers.
<option name=client.engine.timeout	Length of time in minutes that UCF client runs after disconnecting. Longer times mean processes are reused on subsequent logins but more processes will be used on the application server. Since 6.5: Default is 240 minutes.
<option name="ucf.file.buffer.size"	Sets the buffer size (in bytes) used by UCF during transport and file write operations. Default is 32768 bytes. A setting of 61440 may be better for network disk write operations.
<option name="max.parallel.download.streams"	Specifies the maximum number of streams that an individual UCF client can consume. Default is 5.
<option name="min.parallel.segment.size"	Specifies the size (in bytes) of the smallest segment that can be split into multiple streams. A file that is smaller than the specified size is not segmented into multiple streams. Furthermore (with a large file) if the final segment would be smaller than the specified size, it is not split—instead, the previous request would continue reading the remaining bytes of the file. The default is 131072 (128Kb).
<option name="measurement.time.interval"	Specifies the interval (in milliseconds) for the single.thread.throughput option.

Element	Description
<option name="single.thread.throughput">	Specifies the threshold (in bytes) for content transfer throughput on a single thread. If the number of bytes transferred for the interval—beginning with the start of content transfer—specified in the <code>measurement.time.interval</code> option drops below the specified threshold, parallel streaming is turned on. For example, if <code>single.thread.throughput</code> is set to 1048576 (1Mb) and <code>measurement.time.interval</code> is set to 500, the UCF client turns on parallel streaming when less than 1Mb has been transferred in the first 500ms of the content transfer operation.
<option name="progress.notification.period">	Sets the interval (in seconds) for which notification from the UCF client to the UCF server is to occur. Default is 5 seconds.
<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

Table 7. Settings in `ucf.client.config.xml`

<option name="progress.notification.period">	Sets the interval (in seconds) for which notification from the UCF client to the UCF server is to occur. Default is 5 seconds.
--	--

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.

Installing executables with UCF

You can specify one or more executables to be installed by the UCF installer within the optional `<executables>` element of `ucf.installer.config.xml`. In Webtop, the executable files must be placed in the `app\wdk\contentXfer` directory.

In DFC Web applications, you must place these files in any location on the application path that is directly accessible by clients, for example, *root_directory/contextXfer*.

The href attribute value is relative to the contextXfer directory. The installer will download the executable to the client system and execute it from the UCF client installation directory using the specified options.

The attributes are the same as for "lib" and "jar", with two additional attributes: installOptions, and uninstallOptions. Attribute values that contain double quotes must be bounded by single quotes as shown in the following example, and attribute values that contain single quotes must be bounded by double quotes.

```
<executables>
  <executable version="5.3.0.419"
    href="ExMRE.exe"
    installOptions='/s /v"/qn ADDLOCAL=ALL"'
    uninstallOptions='/x /s /v"/qn REBOOT=REALLYSUPPRESS"'/>
</executables>
```

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 an installed WDK application.

In DFC Web applications, you must place this file in any location on the application path that is directly accessible by clients, for example, *root_directory/contextXfer*.

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

To configure the application server to use an SSL certificate that is issued by a certifying authority (CA) that is not trusted by Java or to use a self-signed certificate, do one of the following:

- 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 certificate is placed on the application server, and 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 an installed WDK application. In DFC Web applications, you must place this file in any location on the application path that is directly accessible by clients, for example, `root_directory/contextXfer`. 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 (keystore) 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-based applications in the directory `wdk/contentXfer`.

Tip: In the case of clients that use network shared locations, you must delete the `<ucfHome>` directory that was created on the local client and set the default location to a valid environment variable in `ucf.installer.config.xml`.

In DFC Web applications, you must place this file in any location on the application path that is directly accessible by clients, for example, `root_directory/contextXfer`. 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 in the example.

Encrypting a trust store password — To access the certificates trust store, UCF must have the trust store password. The default password 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_]
```

The classpath (cp) that you provide to this utility must contain references to `ucf-client-impl.jar` and `ucf-client-api.jar`. These APIs are present in the WDK application directory `wdk/contentXfer`. Non-WDK applications must place these APIs within the `WEB-INF/lib` directory.

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 (e.g. https.truststore.password.): UTF-8
```

Copy the encrypted password and encoding class to their respective <option> elements in ucf.installer.config.xml:

```
<option name=https.truststore.password>VIGQdGy1YAQ</option>
<option name=https.truststore.password.encoding>VIGQdGy1YAQ</option>
```

Note: Some algorithms that are listed in Appendix A of the Java Cryptography Extension Reference Guide 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.

Windows client registry in UCF

If UCF is configured to use Windows registry mode, content transfer services use Windows registry keys to record the name and full path to viewed or checked out files and to read the locations of these files. The settings in these keys override the default settings in the UCF client configuration files ucf.installer.config.xml.

If Windows registry mode is configured, and the user does not have the registry keys on the local machine, the default settings from the client configuration file ucf.installer.config.xml are used to create the client configuration locations in ucf.client.config.xml on the client machine. This file is created by UCF in the UCF folder, located in *USER_HOME/username/Documentum/ucf*.

The client application can expose these locations as user preferences. For example, Webtop allows users to set location preferences, which then override both the default configuration settings and the registry settings.

Table 8, page 61 describes the client registry keys. 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 UCF on the client](#), page 52 for details on the client defaults.

Table 8. Content transfer registry keys used by Windows registry mode UCF content transfer

Key Name	String	Purpose
\Common	CheckoutDirectory	Path to checkout directory on client. If not found, defaults to path specified in UCF client config file.
\Common	ExportDirectory	Path to directory for exported and viewed files on client. If not found, defaults to path specified in UCF client config file.

Key Name	String	Purpose
\Common\ViewFiles		Specifies the path to a file that is a file downloaded for viewing and other information.
\Common\WorkingFiles		Specifies the path to a file that is checked out to the checkout directory

Configuring 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.

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` in Webtop applications.

In DFC Web applications, you must place this file in `WEB-INF/classes`. In WDK-based WAR files, this file is located in `WEB-INF/classes`. [Table 9, page 62](#) describes server configuration options in `ucf.server.config.xml`. (If an option does not appear in your file, add it to the configuration.)

Table 9. UCF application server configuration settings

Element	Description
<code>temp.working.dir</code>	Location for temporary upload download of reads and writes, cleaned up content transfer has completed. Not used by WDK, which uses <code><server>.<contentlocation></code> in <code>app.xml</code> .
<code>tracing.enabled</code>	Turns on server UCF tracing to a DFC log file <code>trace.log</code> in the location specified by <code>log4j.properties</code>
<code>file.poll.interval</code>	Interval in seconds (non-negative) to poll for changes to UCF configuration files
<code>compression.exclusion.formats</code>	Specifies formats to be excluded from compression during file transfer.
<code>server.session.timeout.seconds</code>	Sets a timeout that allows UCF servlet to error out if it has not received a request from the application within the configured time. Prevents UCF from being tied up with hung threads.

Element	Description
http11.chunked.transfer.encoding	Sets chunked content transfer 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 Configuring UCF support for proxy servers, page 63 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 Configuring UCF support for proxy servers, page 63 for information.

Configuring UCF support for proxy servers

Some environments with external web servers that serve 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).

Enabling login and session options

The `app.xml` file in each application layer (`wdk/app.xml`, `webcomponent/app.xml`, `webtop/app.xml`, and `custom/app.xml`), contains settings that configure the following login and session options. The following topics describe login and session configuration.

Configuring authentication options

The <authentication> element in wdk/app.xml contains elements whose values are used for login: domain, repository, authentication service class, and single sign-on (SSO). Copy this element to app.xml in /custom and make your changes there. For more information on configuring single sign-on, refer to *Web Development Kit and Webtop Deployment Guide*.

Table 10. Authentication elements (<authentication>)

Element	Description
<docbase>	Specifies 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>	Specifies Windows network domain name
<service_class>	Specifies fully qualified name of class that provides authentication service. This class can perform pre- or post-processing of authentication.
<sso_config>	Contains single sign-on authentication configuration elements
<sso_config>. <ecs_plug_in>	Specifies name of the Content Server authentication plugin (not the authentication scheme name). Valid values: dm_netegrity (for CA) dm_rsa
<sso_config>. <ticket_cookie>	Specifies name of vendor-specific cookie that holds the sign-on ticket, for example, SMSESSION (for CA) CTSESSION (for RSA)
<sso_config>. <user_header>	Specifies name of vendor-specific header that holds the username. RSA value: HTTP_CT_REMOTE_USER. CA value: 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, use SMUSER, if true, use SM-USER.

Supporting saved credentials

The <save_credential> element in wdk/app.xml contains elements that are used for saving user's credentials (username and password) for a repository. Copy this element to app.xml in /custom and make your changes.

Table 11. Save credentials elements (<save_credentials>)

Element	Description
<enabled>	Set to true to enable saved credentials
<encryption_key>	Leave blank. (Provided for backward compatibility.) Set up secure keystore instead (see next paragraph).
<disabled_docbases>	Specifies repositories that will not support saved credentials

To use triple DES encryption, set up a secure keystore location on the file system and specify its location in KeystoreCredentials.properties located in WEB-INF/classes/com/documentum/web/formext/session. You must also override the default keystore location by setting the use_dfc_config_dir key value to false, for example:

```
keystore.file.location=C:/Documentum/config/wdk.keystore
use_dfc_config_dir=false
```

By default, the keystore file location is created in the DFC config directory, which contains dfc.properties and is specified as the value of dfc.config.dir in dfc.properties. The default location is WEB-INF/classes.

Setting trusted domains to prevent redirects

You can prevent redirects outside the application server with a configuration setting in app.xml. Redirects are used by WDK applications in many instances to jump, nest or return to a page within the application, but they can expose the application to security attacks. Add the following elements to custom/app.xml, specifying your trusted application server or servers.

Table 12. Trusted domains element (<trusted-domains>)

Element	Description
<check-redirect>	Set to true to check redirects and deny any outside the trusted domain

Element	Description
<trusted-domain>	One or more elements whose value is a host name, a fully qualified host name, or a host IP address
<error-page>	Relative path to an error page that will be displayed when the URL attempts to redirect outside the trusted domain. The path is relative to the root of the web application and must begin with a forward slash, for example, /custom/errormessage/urlRedirectErrorPage.jsp
<validate-pattern>	Validate patterns (for example, ftp:// or http://) in a relative URL.

Specifying supported languages

The initial locale for the UI presentation at login is determined by the locale of the Java EE server host. The login component presents a language drop-down control that lists all of the supported locales for the application. When the user selects the locale, the UI is refreshed with the UI strings in the language of the selected locale.

In wdk/app.xml, the <language> element contains elements that set the supported locales, default locale, and fallback language. To add a supported locale, copy the <language> element from WDK app.xml to custom/app.xml and make your changes within this element.

Table 13. 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>	Set the righttoleft attribute to true if the locale displays content from right to left. Default="false". 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 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.

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, set this to false to identify non-localized strings, which will be displayed as <code>xxNLS_IDxx</code> where <code>NLS_ID</code> is the NLS key that does not have a value in the specified locale.

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, 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.

Configuring Java EE principal authentication

Java EE principal-based authentication allows a single login to the web server and the Content Server. Each Java EE-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 Java EE principal authentication

1. Make sure that Java EE 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 the file `TrustedAuthenticatorCredentials.properties` located in `WEB-INF/classes/com/documentum/web/formext/session`. To encrypt the password, follow the procedure in [Encrypting passwords, page 69](#).
3. Set up Java EE principals in the application deployment description `web.xml` and in application server-specific files. [To set up Java EE principals, page 68](#) describes how to modify `web.xml`.
4. Stop and restart the application server to enable Java EE authentication.

In a portal environment, user principal authentication requires that the user log on to the portal. The portal username must match the repository username, 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 WDK framework uses the Content Server ticketing mechanism to obtain a ticket for a superuser. The actual username, 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.

Note: With ticketed login, both its creation time and expiration time are recorded as UTC time. This ensures that problems do not arise from tickets used across time zones. When a ticket is sent to a server other than the server that generated the ticket, the receiving server tolerates up to a three minute difference in time to allow for minor differences in machine clock time across host machines. System administrators must ensure that the machine clocks on host machines on which applications and repositories reside are set as closely as possible to the correct time.

To set up Java EE principals

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 Java EE server procedure for setting up Java EE principals. Each Java EE server has its own procedure. For example, in WebLogic, 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 Java EE principals in a configuration file, /conf/tomcat-users.xml.

Encrypting passwords

You can encrypt a password for the repository with the trusted authenticator tool (`com.documentum.web.formext.session.TrustedAuthenticatorTool`). This tool uses triple DES for encryption, generating a key that is stored in a secure keystore location. Use the tool to encrypt passwords for the `drl`, `drlauthenticate`, and `virtuallinkconnect` components. Triple DES encryption is used to encrypt the user's password when saved credentials are enabled in `app.xml`.

You can also use this tool to encrypt passwords for the preferences repository user (`dmc_wdk_preferences`) and the presets repository user in `app.xml`.

To use the password encryption tool

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 path to the class and the repository password to be encrypted:

```
java -classpath "%CLASSPATH%;path_to_WEB-INF/classes;
    path_to_WEB-INF/classes/dfc.jar;path_to_WEB-INF/classes/commons-io-1.2.jar"
    TrustedAuthenticatorTool password
```

The output will look similar to the following:

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

2. For Java EE principal authentication:

Paste the encrypted form of the password into the file `TrustedAuthenticatorCredentials.properties` located in `WEB-INF/classes/com/documentum/web/formext/session`. Each repository must have an entry for the superuser, encrypted password, and domain if needed. Substitute the actual repository name in the sample entries below:

```
Repository_name.user
Repository_name.new-pw
Repository_name.domain
```

If no domain is needed for login, then type the following:

```
Repository_name.domain=
```

For example:

```
mydochase.user=superuser1
mydochase.new-pw=d7d1d6e383d6d4e1d0
mydochase.domain=
```

3. For preferences or presets repository passwords:

Paste the encrypted form of the password into the file `app.xml` in the custom directory. Insert the encrypted preferences password into `<preferencesrepository>.<password>` or the encrypted presets password into the `<presets>.<password>`.
4. The symmetric keys for encryption and decryption are stored in a file named `wdk.keystore`. This file must be stored in a secure location on the application server file system. Open the file `KeystoreCredentials.properties`, located in

WEB-INF/classes/com/documentum/web/formext/session, and specify your keystore location. You must also override the use of the default DFC config dir in order to substitute this new location, for example:

```
keystore.file.location=C:/Documentum/config/wdk.keystore
use_dfc_config_dir=false
```

By default, the keystore file location is created in the DFC config directory, which contains dfc.properties and is specified as the value of dfc.config.dir in dfc.properties. The default location is WEB-INF/classes.

Note: Entries that were encrypted by the 5.3.x encryption tool and entered into the field .password instead of the .new-pw field will be decrypted.

Configuring theme availability

The <themes> element in wdk/app.xml contains elements that define the themes available in the preferences UI.

Table 14. Theme elements (<themes>)

Element	Description
<default_theme>	Specifies the default theme at application startup
<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.

Enabling accessibility (Section 508)

The <accessibility> element in wdk/app.xml contains elements that turn on default accessibility settings (section 508 compliance) in the application. User preference at login overrides the values of this element. Copy this element and its contents to custom/app.xml and make your change to turn on accessibility by default.

Table 15. 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. 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 filename. 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.
<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

For more information on accessibility features in WDK, refer to [Making an application accessible, page 453](#).

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="">
```

Supporting roles and client capability

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. The client capability role model has four fixed roles: consumer, contributor, coordinator, and administrator. User either role model or both in your application.

Using Content Server roles

The <rolemodel> element in wdk/app.xml configures role behavior in the application. The Docbase role model is the default rolemodel class for WDK client applications. This class queries the repository for a list of the current user's roles and for the super roles of the user's roles. The cache of repository roles, their hierarchy, and the user's assigned roles is dynamically every ten minutes. The cache is refreshed when members are added to or removed from a role.

You can specify other types of group classes within the <rolemodel> element to be evaluated by the role qualifier. For example, if your application has a value of mycustomclass for the non-role dm_group value in the repository, you can support this group by adding the class to the <rolemodel> element similar to the following:

```
<groupclasses>
  <group_class>mygroupclass</group_class>
</groupclasses>
```

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. You can specify the domains that are supported by your application in the <rolemodel>.<domain> setting in custom/app.xml. If no domains are specified, the default gets roles in any domain in the repository.

Using client capability roles

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:

Table 16. Client capability mapping to dm_user

Role	client_capability value
consumer	1
contributor	2
coordinator	4
administrator	8

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 17. 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	X
Check out	X	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	*	*	*	*
Perform lifecycle operations	*	*	*	*
Participate in router/ workflow	*	*	*	*
View, edit, or check out workflow			X	X
Create workflow template			X	X
Add to clipboard		X	X	X
Access Webtop admin node				X

* This operation is not scoped by client capability role. Some operations 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.

Using both Server roles and client capability

If roles are not specified as special groups in the repository, 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. Custom roles can be mapped to client capability in the `<client_capability_aliases>` setting so you can use both custom roles and client capability roles in your application. For more information about the user client capability attribute or user role groups, refer to *Content Server Administration Guide*.

The element `<client_capability_fallback_enabled>` specifies whether or not to fall back to client capability if the user does not have a role. If you disable client capability fallback in `app.xml`, the user will have only the roles that are set up in the repository. 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.

The `<fallback_identity>` element in `wdk/app.xml` turns on the DFC fallback identity feature `IDfSessionManager.setIdentity()`. This flag is enabled by default. If your custom component calls `getSession()`, fallback identity 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.

Configuring high availability support

The `<failover>` element in `wdk/app.xml` enables application failover. Set `<failover>.<enabled>` to true to turn on serialization for all failover-enabled components and sessions in the application. By default, failover is enabled. If your custom classes do not support failover, disable failover in your custom component definition or globally in your custom `app.xml` file.

Configuring timeout and number of sessions

Repository session count and timeout — The number of sessions (`dfc.session.max_count`) is configured in the `dfc.properties` file on the Java EE 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.

Web application timeout — The timeout of your web application is managed through the Java EE server. The Java EE 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>
```

Number of sessions — 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. The `<session_config>` element in `wdk/app.xml` contains session management settings for maximum number of application server settings and timeout. Copy this element and its contents to `custom/app.xml` to make your changes:

Table 18. Session management elements (<session_config>)

Element	Description
<code><max_sessions></code>	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 <code>wdk/serverBusy.jsp</code> . A value of 1 means that there is no limit to the number of sessions.
<code><timeout_control></code>	Forces timeout of HTTP session
<code><client_shutdown_session_timeout></code>	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 <code>web.xml</code> , the session timeout will not be overridden.

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 four hours to complete, increase this value to 240. This property is found in the file `FormProcessorProp.properties` located in `WEB-INF/classes/com/documentum/web/form`.

Virtual document operation timeout — The `<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 19. 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.

Troubleshooting timeouts — If the user modifies the URL to go to a component, the session will time out in one minute because the top frame containing the session timeout control has been unloaded. For example, if the user changes the URL from `http://server/webtop` to `http://server/webtop/component/mycomponent`, the session will time out in one minute or less.

Enabling and configuring application-wide features

The following topics describe how to enable or disable specific features in the application.

For information on enabling the Adobe commenting and PDF Annotation Services, refer to [Enabling PDF Annotation Services, page 103](#).

Enabling retention of folder structure and objects on export



Caution: If your users will be exporting folders with special characters (`\ / : * ? " < > |`) in its name, you should not turn this feature on. When a folder with a name containing a special character is encountered during export, an error occurs and the export fails.

To enable retaining the same folder structure (as the one in the repository) and the contained objects on the local file system when the parent folder is exported, add the following element to your `app.xml` in the custom directory:

```
<deepexport>
  <enabled>true</enabled>
</deepexport>
```

The default is false.

Enabling modal pop-up windows

The modal pop-up window is a secondary browser pop-up window with no browser controls either to maximize or minimize the window. This pop-up window appears centered in the screen. The pop-up window provides a similar experience on the web as in desktop, where you can interact with a component in a pop-up window. The user interface for the component appears in a pop-up window (child window) on top of the parent window. If you invoke another component from the child window, the user interface of the component appears on top of the child window and thus stacked one over the other pop-up windows. You cannot access the parent window until you close all the pop-up windows.

The modal pop-up window is supported only on Internet Explorer browser environment. The pop-up window is not 508-compliant and hence it is not supported when 508 accessibility features are turned on through the User Preferences.

In the `wdk/app.xml` file, `<modalpopup>` enables and disables the modal pop-up feature.

[Table 20, page 77](#) describes the elements that configure modal windows in `app.xml`:

Table 20. Modal window elements in `app.xml` (`<modalpopup>`)

Element	Description
<code><enabled></code>	Turns on or off modal windows in the application. Valid values: <code>true</code> <code>false</code> . Default is <code>true</code> .
<code><actioninvocationpostprocessors></code>	List of action invocation post processors specified in <code><postprocessor></code> elements.
<code><postprocessor></code>	Specifies a post processor. The syntax is: <pre><postprocessor id="uniqueId" action="yourAction" class= "YourActionInvocationPostProcessor"/></pre> where <code>uniqueId</code> is an application-wide unique string identifier for the post processor; <code>yourAction</code> (optional) is the name of the action; and <code>YourActionInvocationPostProcessor</code> is the post processor's fully qualified Java class.

Configuring email import

You can configure the application to import email messages with the `.msg` extension as the `dm_message_archive` type with a `.emcmf` extension. The conversion to this type takes place on import.

The emfimport component extends the import component and sets various email attributes on the imported email.

A separate migration utility is available to migrate existing messages of dm_email_message type or subtype to dm_message_archive or a custom subtype. You should run this utility once for each repository that contains saved email messages.

Enabling the EMCMF format in WDK-based applications

You need to install these DARs and DocApps to enable message archive support:

- On 6.5 content servers, install the 6.5 collaborative services-related DARs.
- On 5.3 and 6.0 SPx content servers, install the DCO, collaborative services, and messaging application (which is part of Email Archive) Docapps.

Enabling this feature increases the UCF initialization time, so the first content transfer operation may take some time.

To enable importing email messages in the EMCMF format:

1. Install these DARs and DocApps:

- On 6.5 repositories, install the 6.5 Collaborative Services DARs.



Caution: Webtop 5.3 will not be able to access repositories in which 6.5 Collaborative Services DARs are installed.

- On 5.3 and 6.0 repositories, install the DCO, Collaborative Services and messaging application DocApps.

2. Add the following lines to app.xml (located in the custom directory).

Note: Refer to *Web Development Kit Development Guide* for details on these configuration elements.

```
<messageArchive-support>
  <enabled>true</enabled>
  <supported-file-format>
    <file-format>msg</file-format>
  </supported-file-format>
  <default-attachment-object-type>dm_document</default-attachment-object-type>
  <store-emf-object-as-archive>false</store-emf-object-as-archive>
  <skip-duplicate-messages>true</skip-duplicate-messages>
</messageArchive-support>
```

3. Uncomment the ExMRE line in the wdk/contentXfer/ucf.installer.config.xml file.

<messageArchive-support> configuration elements

Configure the following settings within the <messageArchive-support> element in your custom app.xml:

Table 21. Email conversion settings (<messageArchive-support>)

Element	Description
<enabled>	Set to true to enable conversion. Default: false.
<supported-file-format>	Contains one or more formats (<file-format> elements) of files to be converted to the emcmf format.
<file-format>	Specifies the format of an email file to be converted. Default: msg.
<default-attachment-object-type>	Sets the default type for email attachments. The type can be changed by the user.
<store-emf-object-as-archive>	Specifies whether the converted object is to be stored in archive mode (true) or collaboration mode (false). Default: false.
<skip-duplicate-messages>	Specifies whether to log errors for duplicate messages and continue importing (true) or throw an error and stop importing (false). Default: true.

Log messages

These messages are logged:

- The message could not be imported because email format configuration is disabled.
Reason: The <messageArchive-support> parameter is not enabled and a user attempted to import a .msg file.
- Import of email messages is not supported in HTTP mode.
Reason: A user attempted to import a .msg file in HTTP content transfer mode.
- The file could not be imported as a dm_message_archive object type because it is not a valid file format.
Reason: A user attempted to import, as dm_message_archive, a document other than the supported .msg file.

- The message could not be imported because the selected object type is not a subtype of `dm_message_archive`.

Reason: A user selected an object type that is not a `dm_message_archive` (or one of its subtypes).

- The message "%1" could not be imported because it already exists in this folder.

Reason: If in `app.xml` the `<messageArchive-support>.<skip-duplicate-messages>` is set to `true` and one of these conditions occurred:

- A user attempted to import a message that already exists in the destination folder (that is, the folder from which the user initiated the import action).
 - A user attempts to import a message that already exists in the repository.
- Conversion of mail message failed

Reason: The conversion of a mail message can fail for various reasons, including:

- The Microsoft Outlook client is not installed on client machine.
- The EMC EmailXtender MRE utility (`ExMRE.exe`) is not installed on client machine.
- The mail message file is not valid.

Enabling shortcuts (hotkeys)

Shortcuts, also called hotkeys, are key combinations that can be configured to call WDK actions or to invoke UI controls such as buttons or links. Shortcuts are enabled for the entire application in `app.xml`. To disable hotkeys, add the following to a modifications file, for example, `custom/app_modifications.xml`.

```
<application modifies="wdk/app.xml">
<replace path="hotkeys">
  <hotkeys>
    <enabled>false</enabled>
  </hotkeys>
</replace>
```

Note: Because the `<application>` element that you are modifying does not have an ID, you must leave it out of the `modifies` attribute value as shown in the example.

The `<nlsbundle>` element refers to a Java properties file. This is a text file that creates the mapping for your hotkeys. If you add or change hotkeys, specify your own properties file as the value of `<nlsbundle>`.

For information on configuring shortcut combinations for individual controls in the application, refer to [Configuring and customizing shortcuts to actions \(hotkeys\)](#), page 222.

Enabling datagrid features

Datagrids support mouse or keyboard row selection (one or more objects), right-click context menus, fixed column headers, and resizable columns. These features can be disabled for the application in the app.xml element <desktopui>. Add the following element to your app.xml in the custom directory to disable row selection:

```
<desktopui>
  <datagrid>
    <rich-ui>>false</rich-ui>
  </datagrid>
</desktopui>
```

For information on configuring row selection on individual datagrids, refer to [Configuring datagrid row selection, page 188](#). For information on configuring right-click menus for selected objects, refer to [Configuring context \(right-click\) menus, page 134](#). For information on configuring fixed column headers and resizable columns on individual datagrids, refer to [Configuring column headers and resizing, page 186](#).

Enabling autocompletion

Autocompletion for string input controls can be enabled or disabled in wdk/app.xml, in the <auto_complete> element. Application-wide support for auto-completion is configured in the <auto_complete> element. Users can also enable or disable autocompletion globally in their general preferences settings and clear their autocompletion lists. If autocompletion is disabled globally in app.xml, users cannot enable it.

[Table 22, page 81](#) describes the elements that configure autocompletion in app.xml.

Table 22. Autocomplete elements in app.xml (<auto_complete>)

Element	Description
<enabled>	Turns on or off autocompletion in the application. Valid values: true false
<maxentries>	Specifies the maximum number of entries to be stored for the autocompleteid. Default = 200
<maxsuggestions>	Specifies the number of matching suggestions to display to the user. Default = 10

For information on configuring autocompletion, refer to [Configuring autocompletion, page 134](#). For information on how to add autocompletion support to a custom control, refer to [Adding autocompletion support to a control, page 161](#)

Configuring presets access

By default, presets are stored in the global repository unless a presets repository is configured in app.xml. Presets are available only for Content Server 6 repositories because they rely on types that are created by the presets DocApp.

Table 23, page 82 describes the entries that configure presets behavior in the application. For information on using presets in a custom component, refer to [Using custom presets, page 343](#).

Table 23. Presets elements (<presets>)

Element	Description
<password>	Encrypted password for the default preset user, dmc_wdk_presets_owner. If a value is not specified, the default password is used. To create an encrypted password, execute this command at a command prompt: <code>java com.documentum.web.formext.session.TrustedAuthenticatorTool password</code> The encrypted password is sent to standard output.
<refresh_period_seconds>	Interval to check for preset package changes in the repository. The beginning of the interval is the actual execution time of the last refresh.
<enabled>	Set to true to enable presets in the application
<version>	Version of preset package in DocApp. Must match DocApp.
<repository>	Name of repository in which to store presets. If element is empty or missing, presets are stored in the global registry repository.
<repository_path>	Path in repository to presets storage folder.
<refresh_period_seconds>	Interval at which to check for change to presets in repository.
<scope_name_mappings>	Contains one or more <scope_name_mapping> element.
<scope_name_mapping>	Optional mapping of a scope name in the WDK application to a preset scope
<preset_scope_definition_name>	Name of preset scope in preset package
<qualifier_scope_name>	Name of qualifier defined in app.xml

To give users the ability to create presets using the presets editor, assign those users the role `dmc_wdk_presets_coordinator`.

Configuring a preference repository and preference cookies

You can configure a repository to store user preferences that should be stored. Preference storage in the repository enables users to see their preferences on more than one machine.

If no preference repository is configured in `app.xml`, the global repository is used to store user preferences. User preferences that should not be stored (cookie only) should be added to the `<non_repository_preferences>` element.

This section describes the `app.xml` configuration. For more information on configuring and customizing user preferences and managing cookies, refer to [Configuring and customizing preferences, page 329](#).

[Table 24, page 83](#) describes the elements within `app.xml` that register a user preference storage class and configure a preference repository.

Table 24. Preferences configuration elements

Element	Description
<code><environment>.<serverenv><preferencestore-class></code>	Specifies the fully qualified name of a class that instantiates a class to store user preferences.
<code><preferencesrepository></code>	Contains a <code><repository></code> element. If this element is not present, user preferences are stored in the global repository, which can slow down performance.
<code>.<repository_path></code>	Specifies the path within the preference repository in which to store preferences. If the path does not exist at application startup, it will be created.
<code>.<repository></code>	Specifies the repository in which to store preferences, preferably not the global repository.
<code><non_repository_preferences></code>	Contains preferences that should not be stored in a repository and should be stored only in a cookie, such as login preferences, which are used before the preferences are downloaded from the preferences storage repository.

Element	Description
<preference>	Contains the XML path to the element within a configuration file that defines the cookie (nonrepository) preference. If an element has an ID, it must be specified, as in the following example from the login component <username> element: component[id=login].username.
<password>	<p>Encrypted password for the default preset user, dmc_wdk_presets_owner. If a value is not specified, the default password is used. To create an encrypted password, execute this command at a command prompt: <code>java com.documentum.web.formext.session.TrustedAuthenticatorTool password</code></p> <p>The encrypted password is sent to standard output.</p>

Add the user credentials for a user that has read and write privileges in the preference repository to the TrustedAuthenticator.properties file in WEB-INF/classes/com/documentum/web/formext/session. Use the Trusted Authenticator tool to encrypt the user’s password. For instructions on how to use this tool, refer to [To use the password encryption tool, page 69](#).

Enabling and disabling session state

To enable or disable session state for a client, set the <client-sessionstate> element. You can use a <filter> and a corresponding clientenv attribute value to specify the client.

Enabling redirect security

Redirects are used by WDK-based applications in many instances to jump, nest or return to a page within the application, but they can expose the application to security attacks. Prevent redirects outside the application server with a configuration in app.xml. Add the following elements to custom/app.xml and specify your trusted application server or servers. The value of <trusted-domain> can be either a hostname, a fully qualified hostname, or host IP address as shown in three different examples:

```
<trusted-domains>
  <check-redirect>true</check-redirect>
  <trusted-domain>66.94.234.13</trusted-domain>
  <trusted-domain>localhost</trusted-domain>
  <trusted-domain>myserver.mycompany.com</trusted-domain>
</error-page>/wdk/system/errormessage/urlRedirectErrorPage.jsp</error-page>
```

```
</trusted-domains>
```

Enabling the streamline view

The streamline view, present in 5.3.x, is deprecated in version and turned off by default. To turn it on, set `<streamlineviewvisible>` to true in your custom app.xml file.

Enabling notifications

Two kinds of notifications are enabled in webcomponent/app.xml: event notification and read notification. Notification requires a valid SMTP server configured for the Content Server: `smtp_server_attribute` of `dm_server_config` object. The user who requests notification as well as the user whose activity generates a notification must each have a valid email address configured in the `dm_user` object.

Change notification — Users can request change notification for a Content Server event on one or more objects. Any API, workflow, or lifecycle event can be notified. Notification is available on `dm_sysobject` and its subtypes from the Webtop menu or the right-click context menu. Notification on replica and reference (shortcut) objects is not supported. The user who has selected change notification on an object will receive a notification in the Documentum inbox, and by email. If an event that is configured in app.xml does not exist in a particular repository, that event is ignored by the event notification mechanism for users who are logged into that repository.

Read notification — A user can select any object in a list view and turn on read notification to notify the user when the document has been read. The notification will contain the object name, title if present, last modified date, repository location, and name of the user who read the email. The user who selects read notification must have at least read permission on the document or a more restrictive permission if it is configured in the `<readnotification>` element. The read notification is triggered by a Content Server `dm_getfile` event, which includes view, edit, export, and checkout.

If your application has customized the Content Server email script, add your custom content to the Content Server 6 scripts `dm_event_sender.ebs` and/or `dm_html_sender.ebs` located at `$DOCUMENTUM/product/version_number/bin` on the Content Server.

[Table 25, page 85](#) describes the elements in webcomponent/app.xml that configure notifications.

Table 25. Event notification elements (<notification>)

Element	Description
<code><notification></code>	Contains events for which users can be notified. Contains one <code><events></code> element.

Element	Description
<notification>.<events>	Contains one or more <event> elements that will be made available to users who select notification
<events>.<event>	Specifies the name of a Content Server API event
<readnotification>	Enables notification of file access (read events). Contains one <event> element and a <minpermit> element.
<readnotification>.<event>	Specifies the name of a Content Server API event
<readnotification>.<minpermit>	Specifies the minimum permission on the selected object that is required for event notification

Enabling 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.

To find out which components support failover, check the component configuration file for the value of true in the element <failoverenabled>. Refer to [Implementing failover support, page 292](#) for information on implementing failover in custom components.

The following topics describe failover configuration.

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 failover can be disabled for environments that do not support failover such as portals. If you migrate WDK 5.3 customizations to WDK 6, and your custom classes do not support failover, disable failover in your custom app.xml file or 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 292](#).

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 anonymous virtual link access

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

1. Credentials passed with the request
2. Current WDK session
3. Virtual link anonymous account
4. 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.

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.

To set 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 69](#)).
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 and Webtop Reference* for information on configuring the `virtuallinkconnect` component.

Virtual links overview

A virtual link is a URL to view a single document and 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.

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

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

or

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

Virtual links are resolved to a matching object in the named repository for the named application. To handle URLs without the application virtual directory, such as `http://myhost/somecabinet/somedoc`, deploy virtual link support to a default WDK application. For instructions, refer to *Web Development Kit and Webtop Deployment Guide*.

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.

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 filename 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
```

If there is no match, the handler will remove the extension and look for the object.

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. 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.

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 were created based on the folder structure of the document inside the repository. For example, if the user views 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. 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 and object name, 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 uses the HTTP content transfer mechanism.

Enabling discussion sharing

The `<discussion>` element in `wdk/app.xml` contains the following elements that configure discussions.

Table 26. Documentum Collaborative Services elements (<discussion>)

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

Configuring hidden objects and invalid actions display

The `<display>` element in `wdk/app.xml` configures the display of hidden objects and invalid actions for the application. Set `<display>.<hiddenobject>` to `false` to display hidden objects. Set `<display>.<hideinvalidactions>` to `true` to hide all invalid actions in the application. This setting overrides the `showifinvalid` attribute valid on individual controls.

Supporting asynchronous jobs

The asynchronous framework in WDK allows component and action jobs to run asynchronously, returning control to the client immediately.

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.

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.

You can add the <job-execution> element and its contained elements to your custom application configuration file to support global settings for asynchronous jobs (actions and components).

Table 27. 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.

Element	Description
<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 [Creating asynchronous jobs, page 364](#).

Configuring applet display

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

Table 28. 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 without the plugin.

Element	Description
<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-001400020008ABCDEFEDCBA 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_2windows-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.

Configuring the copy operation

The <copy_operation> element in wdk/app.xml contains an element that determines whether to retain storage areas during copy operations. Some applications override the default setting for this feature.

Table 29. Copy operation elements

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

Configuring the move operation

The <move_operation> element in wdk/app.xml contains an element that determines whether to move all versions or only the selected version during move operations.

Table 30. Move operation elements

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

Configuring navigation defaults

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 31. Navigation settings

Property	Description
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
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 exceeds the value of <max_sessions> in app.xml
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.

Setting web-link type in Application Connectors

Users of Application Connectors will have a menu item in the Documentum menu bar that inserts a link. By default the link is a DRL. When another user selects it in the document, it will launch Webtop

in the same way that a web link sent by the **Webtop Email as web-link** menu item. The following link is an example of an embedded DRL:

```
http://pleeng0148:8080/webtop/drl/objectId/090000018061c7cf
```

To use a virtual link to a relative folder path instead of a DRL, set the value of <insert_link> within the element <appintg_link> in your custom app.xml to virtual_link. The resulting link in your document will be similar to the following:

```
http://pleeng0148:8080/webtop/dm_notes://Jerzy%20Gruszka/aaaa1.doc
```

The user can select any text within the document to insert the link, and the text will then be underlined and colored blue.

Configuring 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.

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 you can tune memory usage, since memory is consumed by maintaining history for each browser window in each user session.

Turn on browser history management and set the number of page requests that are retained for browser history in the file FormProcessorProp.properties, which is located in WEB-INF/classes/com/documentum/web/form.

 **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

1. Open FormProcessorProp.properties in 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 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.

Registering custom classes

The <qualifiers> element in wdk/app.xml 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.

The <errormessageservice> element in wdk/app.xml registers a class that provides error messages.

The <infomessageservice> element registers a class element for a class that provides informative messages. Register your custom message classes within these elements in custom/app.xml.

The <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.

The <listeners> element registers application, session, and request listeners.

Table 32. 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>	Specifies the fully qualified class name of the listener

The <xforms> element registers an XForms adapter service class.

Table 33. Process Builder Forms Builder elements (<xforms>)

Element	Description
<adaptorService>	Fully qualified class name

Configuring lightweight sysobject parent display

A lightweight sysobject (LWSO) is the child of a parent object that provides metadata such as security, retention, status or storage. The parent shares this metadata with multiple lightweight objects. This reduces disk storage space and speeds ingestion for indexing. You can configure the WDK-based application to display parent object that are shareable in listing and locator components.

Table 34. Displaying lightweight sysobjects (<lightweight-sysobject>)

Element	Description
<hide-shared-parent>	Set to true to hide LWSO parents in listing and locator components. Default or unspecified: true

For information on how to add support for lightweight sysobject parent display in a listing component, refer to [Supporting lightweight sysobject display in a component](#), page 212.

Enabling and configuring drag and drop and spell check

The following topics describe settings in app.xml that configure drag and drop and spell checker installation and behavior.

Configuring the Active-X plugin install

The <plugins> element in wdk/app.xml contains elements that enable the spell-check dictionary and drag and drop Active-X plugin for Internet Explorer. The plugin is located in wdk/native.

Table 35. Active-X plugins elements (<plugins>)

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

Element	Description
<enabled>	Enables 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, the plugin will download at the first drag action, and 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.

Configuring drag and drop

The <dragdrop> element in wdk/app.xml contains elements that turn on or off drag and drop support in the Internet Explorer browser. For information on customizing drag and drop, refer to [Supporting drag and drop, page 298](#).

Note: Drag and drop does not use UCF for content transfer. Drag and drop to or from the desktop will not use the closest ACS or BOCS server to the user and will always transfer content to or from the primary Content Server.

The following components support drag and drop:

Table 36. 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

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

Table 37. 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.

Enabling the rich text spell checker

The <richtexteditor> element in wdk/app.xml contains element that configures the rich text editor.

Table 38. 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.

Configuring file format, rendition, and PDF Annotation Services support

The following topics describe how to map file extensions to formats, set preferred renditions and viewing/editing application, and enable PDF Annotation Services.

Mapping extensions to formats

The `<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 39. Formats elements (`<formats>`)

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

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

Supporting XML file extensions

The <xmlfile_extensions> element in wdk/app.xml contains <extension> elements. These elements configure the file extensions that will be recognized and parsed as XML files by the application. By default, a WDK-based application retrieves the dm_format object based on the object's 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 40. 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.

Specifying preferred renditions and their viewing and editing applications

The <preferred_renditions> element in wdk/app.xml contains elements that specify the default list of renditions (document type and format combinations) and 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 41. Preferred renditions elements

Element	Description
<filter>	Applies the preferred renditions list to the client environment that is specified in the clientenv attribute
<nlsbundle>	Specifies the fully qualified class name for the NLS class that resolves strings used by the preferred renditions service
<renditions>	Contains <rendition> elements that are the default renditions
<rendition>.<mode>	Specifies action to be performed. Valid values: view edit
<rendition>.<object_type>	Specifies the object type in the repository, such as dm_document, or all_types
<rendition>.<primaryformat>	Specifies 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>	Specifies for view mode 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. Cannot be used with an <action> element. If blank, the user's preferred viewing application specified in the user's preferences will be used.
<rendition>.<action>	(Optional) Specifies the name of the action to be invoked, if the rendition invokes an action instead of an application. Use instead of an <app> element.
<rendition>.<inline>	(Optional) Specifies whether the rendition can be displayed inline (view mode only). If set to true, document will be launched with HTTP content transfer mode. If set to false, you can specify a viewing application in the <app> element. If blank, the user's preferred viewing application specified in the user's preferences will be used.

Element	Description
<code><rendition>.<label></code>	(Optional) Specifies the application label to be displayed in the format preferences UI drop-down list. If omitted, the contents of the <code><app></code> tag will be used for the app label.
<code><rendition>.<isdefault></code>	(Optional) Enables 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.

Custom settings should be done in an `app_modifications.xml` file in `custom/config` as shown in the following example:

```
<application modifies="wdk/app.xml">
  <insert path="preferred_renditions.renditions">
    <rendition>
      <mode>view</mode>
      <object_type>dm_document</object_type>
      <primaryformat>msw8</primaryformat>
      <renditionformat>pdf</renditionformat>
    </rendition>
  </insert></application>
```

Enabling PDF Annotation Services

You can enable the use of PDF Annotation Services in any WDK application. The `<adobe_comment_connector>` element in `webcomponent/app.xml` specifies the necessary connection settings 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 42. 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>	Enables using virtual links to retrieve content. Set to false to enable comment on the current document only.

Element	Description
<formats>	Contains all of the formats that can support Adobe comments
<format>	Specifies a format for which to allow Adobe comments, for example, pdf or msw8

Configuring the application environment

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

- Java EE Application servers

The stand-alone application environment supports web applications in a Java EE application server. Authentication and configuration services are provided by the Java EE 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 deployed.

- 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 servers environment and styles.

- Windows content authoring applications

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

The following topics describe how to configure the application environment.

Configuring application 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. You can change these values to use your own classes.

Table 43. Environment settings

Setting	Description
LookupHookPath.#	Specifies the path, scope argument, and class name for each configuration service listener class. Refer to Configuration lookup hooks, page 433 .
LookupHookArgument.#	
LookupHookClass.#	

Setting	Description
SessionHookClass	Specifies a session listener class that implements ISessionHook in com.documentum.web.formext.session
ComponentServletPath	Specifies the path used by Component. getServletPath to resolve the path statically rather than dynamically. Should match the ComponentDispatcher servlet mapping in web.xml.
ConfigReaderClass	Specifies the class that parses XML configuration files
non_docbase_component.#	Specifies the name of a component that does not require a Documentum session (login is not presented)
forward_dispatch_component.#	Specifies the name of a component that requires forward dispatching instead of the default include dispatching between servlets and components. Components that set response headers require forward dispatching.
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.

Specifying supported browsers

The elements within <browserrequirements> in wdk/app.xml enforce supported browsers and platforms and provide strings for the error messages to be displayed when the user has an unsupported environment. Use <nlsid> element within the message elements to render a message that is localized.

Table 44. 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

Element	Description
<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>	Contains an error message to be displayed when the Netscape browser does not allow software installation

Turning on cross-site scripting security

The <requestvalidation> element in wdk/app.xml 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 45. URL request validation elements (<requestvalidation>)

Element	Description
<enabled>	Validates every HTTP request for the parameters named in the <parameter> elements if the value is set to true.

Element	Description
<parameter>	Contains a parameter that must be validated. Must contain a <name> element and at least one of the following child elements: <datatype>, <regex>, <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 regex 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 IRequestParameterValidator.
<regex>	Regular expression that must be satisfied by the parameter value, for example, __client(\d+)(~)(\d+). For information about Apache expression syntax, refer to the Apache website .
datatype	Datatype of the parameter value, for example, String, Integer, Long. String type requires a regular expression (<regex>).

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.

Configuring the client environment

The <environment> element in wdk/app.xml sets environment-specific dispatching of components and other environment-specific settings. The <environment> element contains one <clientenv>, one <clientenv_structure>, and one <serverenv> element.

Table 46. 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 47. Server environment elements (<serverenv>)

Element	Description
<filter>	Specifies a valid client environment. The value of the clientenv attribute must match one of the client environments defined in <environment>.<clientenv>.
.<class>	Specifies the fully qualified name of a class that instantiates the server environment
.<preferencestoreclass>	Specifies the fully qualified name of a class that instantiates a class to store user preferences.

The <client-sessionstate> element contains filters that enable or disable specific client environments.

Table 48. Client session state elements (<client-sessionstate>)

Element	Description
<client-sessionstate>	Contains one or more filters that enable or disable a client environment
<filter>	Enables or disables a client environment. The value of the clientenv attribute must match a client environment specified in <environment>.<clientenv>.
<enabled>	Enables the client environment

Configuring type icon display

To display an icon for an object type rather than the special icon that denotes virtual document or assembly members, configure your custom app.xml for the type. Add the type as a value of <always-show-icon-type> within the <docbaseicon> element. (You must copy the entire element and its contents to add your type.)

Configuring web deployment descriptor settings

A deployment descriptor file is defined and described by the Java EE 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. The following topics describe WDK additions to web.xml.

- [Naming the application, page 110](#)
- [Specifying static pages for better performance, page 110](#)
- [Reducing HTTP sessions, page 111](#)
- [Specifying error pages, page 111](#)
- [WDK servlet filters, page 111](#)
- [WDK servlets, page 112](#)
- [WDK listeners, page 114](#)

Naming the application

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, change the value in web.xml to match the folder name.

Specifying static pages for better performance

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 49. Static page context parameters

Context Parameter	Description
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: <code><!1-opening-square-bracketCDATA1-opening-square-bracket^.*?\formaticon\.\+?\fileExt\.\.*?\file\.gif\$2-closing-square-bracket></code>

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>

```

Reducing HTTP sessions

The context parameter `HTTPSessionRequired` specifies URLs that do not require a new HTTP session in 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 website](#).

Specifying error pages

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 50. <errorpage>

Element	Description
<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 context parameter `UseVirtualLinkErrorPage` HTTP "404 - File Not Found" tries to resolved errors as virtual links. Set to true to use the `VirtualLinkHandler` servlet for 404 errors.

WDK servlet filters

The table [Table 51, page 112](#) describes the servlet filters that are defined in `web.xml`. These filters intercept URLs that match the `url-pattern` value.

Table 51. 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 292 .
RequestAdapter	Processes requests and intercepts requests with multipart/form-data 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 443 .
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 443 .

WDK servlets

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

Table 52. 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

Servlet	Description
UcfNotification	Specifies the servlet that implements INotificationHandler, which UCF uses to notify the client of errors and progress
WorkflowEditorServlet	Specifies the servlet that displays Web Workflow Manager
VirtualJS	Specifies the servlet that 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	Specifies the servlet for tracing in WDK
ComponentDispatcher	Specifies the component dispatcher servlet, which maps a URL to a component to the appropriate component start page.
ActionDispatcher	Specifies the action dispatcher servlet, which maps a URL to an action to launch the action.
DRLDispatcher	Specifies the servlet that converts a DRL to a URL
SessionTimeoutControl	Specifies the servlet that overrides the JSP container timeout to provide finer timeout management
wdk5-download	Specifies the WDK 5 content transfer servlet that streams browser-supported content to the browser.
wdk5-appletresultsink	Specifies the WDK 5 servlet used by content transfer applets to return results
HttpContentSender	Specifies the servlet that wraps HttpSessionServlet
FileFormatIconResolver	Specifies the servlet that returns a format icon for a given file extension
DesktopDragData	Specifies the servlet that gets data on files dragged from the desktop
DownloadServlet	Specifies the servlet that is used by HttpContentTransportManager to stream content to the browser
VirtualLinkHandler	Specifies the servlet 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.
PortletServlet	Specifies the servlet that handles URLs with the with the pattern /portlet/*

Servlet	Description
TestCaseDriver	Specifies the servlet that records or runs test cases and test suites
ImagingServiceServlet	Specifies the servlet that handles imaging service requests

WDK listeners

The following listener is specified in web.xml:

Table 53. Deployment descriptor listener

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

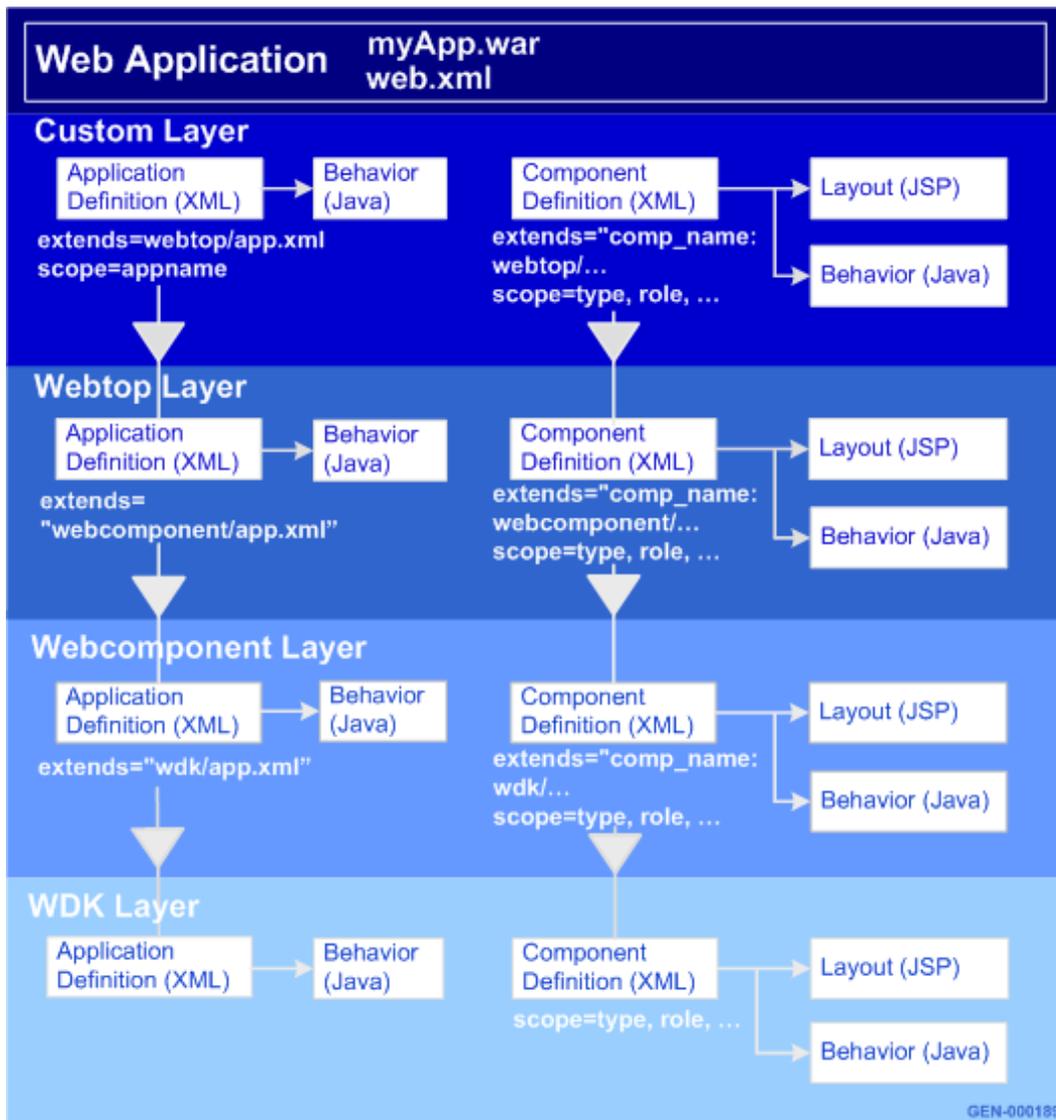
Application layers

The base application layer is wdk, in the wdk directory. This layer is defined in the in app.xml file located in /wdk. The webcomponent application layer is built on top of the wdk layer and configured in the app.xml file located in /webcomponent. The webtop application layer is built on the webcomponent layer. Each application layer adds its own parameters and overrides some of the inherited parameters from parent application layers.

Each application layer within the web application must have an XML configuration file named app.xml at the root level of the application layer. Your custom app.xml file should extend the top application layer. For example, if your application is Web Publisher, the top layer app.xml is in /wp. Place your configuration files, custom JSP pages, and NLS resource files in the custom directory or in subdirectories of custom.

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

Figure 4. Application layers and configuration inheritance



Tip: The /custom or user-defined directory is the top-level application layer. Do not add your custom application files to the wdk, webcomponent, webtop, or other EMC Documentum directories. They will be lost when you upgrade the content of those directories. If you rename the custom directory, you must specify the new directory name in your WEB-INF/web.xml file.

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 the custom application layer, 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 [Modifying or extending a component definition, page 249](#).

To create a custom application layer

1. Add an app.xml file to the custom directory and specify the name of the application that it extends. It must extend the top application layer, for example:

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

2. Create directories for your application-specific components and add the JSP pages and JavaScript files for those components.
3. Add configuration files for your components to the /config directory.
4. 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.
5. (Optional) If you changed the name of the custom directory, specify the custom 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>
```

Contents of an application layer

The WDK framework manages an application based on application layers. For more information on application layers, refer to [Application layers, page 114](#).

The main configuration file and directories in an application layer are the following:

Table 54. Main directories and files in a WDK-based application

Directory or file	Description
app.xml	Application configuration file. Can extend another application's app.xml file. Contains application-wide behavior classes such as qualifiers, servlets, and listeners.
/config	Contains configuration files for the application-layer components

Directory or file	Description
/include	(optional) Contains JavaScript files that are used within the application layer or within applications that extend the application
/src	Contains Java source files for components and, in some cases, controls in that layer
/strings	Contains externalized strings for actions and components in the application layer
/theme	Contains themes (icons and style sheets) for the application

Configuring and Customizing Controls

This chapter addresses common control configuration and customization tasks:

- [Control configuration overview, page 119](#)
- [Configuring controls, page 124](#)
- [Control customization overview, page 146](#)
- [Customizing controls, page 153](#)
- [Programming control events, page 172](#)

Refer to [Chapter 4, Configuring and Customizing Data Access](#) for information on configuring and customization data access controls.

Control configuration overview

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. The control tag class 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.

Use controls for the following purposes:

- Accept user input
- Raise events that change the behavior of the component that contains the control
- Format output

- Change the display of object attributes
- Launch an action
- Bind to and display data
- Display a different set of attributes for each component UI
- Validate user choice

For information on individual controls and their configuration, refer to *Web Development Kit and Webtop Reference*.

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 and Webtop Reference* for more information on this attribute. These tags are defined in the tag library `dmformext_1_0.tld`.

For information on configuring a specific control, refer to *Web Development Kit and Webtop Reference*.

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`, `docsizevalueformatter`, `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 does not have a Documentum session. Refer to [Providing data to databound controls, page 202](#) 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`, and `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.

The `docbaseattribute` tag renders a label and attribute value.

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 more information on configuring attributes and attribute lists, refer to [Generating lists of attributes for display, page 196](#).

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.

You can 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. If your action control has optional nested argument tags, they are passed to the action and to any component that is launched by 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.

The `actionbuttonlist` and `actionlinklist` controls display multiple actions. The actions in these lists are defined in an action XML definition.

An action control can be static or dynamic. The action of a static control is specified at design time by an action attribute on the control in the JSP page. The action is executed when the control action button, link, menu item, or multiple checkboxes are selected. These controls do not support the "dynamic" attribute.

A dynamic action control is evaluated at runtime to determine its visibility and enabled state. This dynamic behavior is configured using the dynamic attribute on the action control. Dynamic action controls do not support the `runatclient` attribute.

multiselect — The value of "multiselect" for the dynamic attribute specifies whether an action can be performed on multiple objects. If a control's dynamic attribute is set to `multiselect`, the controls 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.

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 `format_preferences_summary.jsp` page contains an **Edit** button that is enabled only when one object checkbox is selected, so that only one format preference can be edited at a time:

```
<dmfx:actionlink nlsid="MSG_EDIT" dynamic="singleselect" action="editformatpref" .../>
```

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 55. State of a control based 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: 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>
```

How value assistance is rendered

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 `docheckboxattributevalue` 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, 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"/>
```

Configuring controls

The following topics describe common control configuration tasks and general information about controls.

Finding the file for configuring a control

A feature that is displayed in the UI may be generated by more than one component, so it can be difficult to locate the file in which to configure a control. The contributing components 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 layers, page 114.](#))

To find the file that contains a control:

1. Locate the string in the UI that accesses the feature to be configured.
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:

This example locates the file that contains a button and changes the string as well as the action that is launched by the button.

1. Identify a string in the UI that is associated with the feature of interest. For example, in the Webtop frameset, there is a series of buttons across the top. The task is to launch a custom logout component from the **Logout** button.

- Using a multi-file search tool, search all *.properties files for the string "Logout" in the strings directories of the web application.

The string "Logout" is found in five files. You can eliminate the testbed file and the logoff component, because the button you are searching for is not in the logoff component. Of the three remaining, one is in the menubar component, but you have not opened a menu. Another is in the generic actions properties, and the last and correct string is in the Webtop titlebar component properties file TitleBarNlsProp.properties.

- Search *.xml files for the configuration file that contain this resource bundle. Drop the .properties extension in your search, because the bundle name does not have an extension.

"TitleBarNlsProp" is found in webtop/config/titlebar_component.xml. This is the component you will need to extend, because the logout button on the JSP page must launch your custom component.

- Create a modification XML file with the following content:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<config>
<scope>
<component modifies="webtop/config/titlebar_component.xml">
  <replace path="pages.start">
    <start>custom/titlebar.jsp</start>
  </replace></component></scope></config>
```

This modification file will direct the configuration service to use your titlebar JSP page rather than the Webtop page.

- Copy webtop/titlebar/titlebar.jsp to /custom and open the file in your editor.
- Locate the UI control that launches the logout component. In this example, search on the NLSID key MSG_LOGOUT. You will find this key as the nlsid attribute value for a button tag:

```
<dmfx:actionbutton name='logout' action='logout'...>
```

Since this is an actionbutton, the action launches the logout action. Change this to the name of the action that will launch your custom logout action and add the NLS key for your string, for example:

```
<dmfx:actionbutton name='logout' action='mylogout'...>
```

- Create a simple action definition for your custom logout component, for example:

```
<action id="mylogout">
  ...
  <execution class="com.documentum.web.formext.action.LaunchComponent">
    <component>mylogout</component></class>
</action>
```

- Save your action definition and JSP page and then refresh the configuration files in memory by navigating to wdk/refresh. You can now view the page that contains your customized button or link. Your custom component should be launched when you click the button or link.

Configuring a control

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.

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.



Caution: You must use action controls or controls that get data from a repository (dmfx:...) within a component JSP page. Control tags will not work properly in JSP pages that are not within the component framework.

- `dmwebtop_1_0.tld`
Contains Webtop-specific controls.
- `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

To add a control tag

1. 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" %>
```

2. Add the control tag to 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 this 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 runtime.
- `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.

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 and Webtop Reference*.
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.)

Some controls have additional configuration through an XML configuration file. For information on these controls, refer to [Global control configuration, page 127](#).

Global control configuration

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 and Webtop Reference*.

Table 56. Control global configuration

Control	Configuration
advsearch (various controls)	Configure value assistance (refer to Programmatic search value assistance, page 404), default match case (refer to Search query performance, page 443), size and dates (refer to Configuring search controls, page 389 in wdk/config/advsearchex.xml)
breadcrumb	Configure breadcrumb controls to display or hide last leaf of breadcrumb and its style (refer to breadcrumb in <i>Web Development Kit and Webtop Reference</i>). Set components that should show a relative path in the breadcrumb.
date controls	Configure dateinput, datevalueformatter, and datetime controls (refer to <i>Web Development Kit and Webtop Reference</i>)
docbaseattributelist	Configure the display of attributes webcomponent/config/library/attributes_*_docbaseattributelist.xml (refer to Creating an attributelist that is independent of the data dictionary, page 198)
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 162)
iconwell	Configure (add to) any component definition whose JSP page can display an iconwell (refer to iconwell in <i>Web Development Kit and Webtop Reference</i>)
paneset controls	Configure paneset controls to govern screen real estate. Refer to the paneset control in <i>Web Development Kit and Webtop Reference</i> for details.
richtexteditor, richtextdisplay	Configure allowable HTML tags for user input wdk/config/richtext.xml (refer to Configuring rich text, page 140)
xforms	Customization not supported wdk/config/xforms_config.xml

Configuring tabs

This section describes how to add a tab in a component. The properties component JSP page has a tabbar control that renders a tab for each component listed in the properties component definition. Three tabs are displayed in Webtop based on the following XML entries:

```
<contains>
  <component>attributes</component>
  <component>permissions</component>
  <component>history</component>
</contains>
```

Figure 5, page 129 shows the tabs in Webtop:

Figure 5. Properties tab bar



To add a tab to the properties tab bar

1. In your custom directory, add a new component configuration file, `properties_component.xml`. The component element should modify the Webtop definition:

```
<component modifies="properties:
  webcomponent/config/library/properties/properties_component.xml">
</component>
```

2. Insert a `<component>` element into the `<contains>` element:

```
<insert path="contains">
  <component>versions</component>
</insert>
```

3. Refresh the configuration files in memory by navigating to `wdk/refresh.jsp`. Figure 6, page 129 shows the new tab the next time you view properties:

Figure 6. Custom tab bar



Configuring tab key order

You can specify the order in which tabbing between form fields and other UI elements will occur in the browser and specify which field on a given form receives the focus when user first navigates to that form. For example, a given form may support a business process in which the user wants to tab vertically through multiple fields. A different form may require tabbing horizontally through a set of fields. You may require some UI widgets on a given form to not be in the tab path.

By default, most browsers support tabbing through HTML elements in the order that they appear in the character stream. Browsers generally interpret tab indexes relative to each individual frame, starting at the top left of a frame and moving from left to right and then down to the next line. Elements are tabbed to within a frame and then focus moves to the next frame. In compliance with the HTML 4.01 specification, WDK will generate a `tabindex` attribute on HTML elements that overrides this default behavior. Most browsers will honor a value of 1 as an indication that the element cannot be tabbed to.

Tip: As a general rule, you should assign index values incrementally based on the order of elements in the source code for the page. Use the tab key ordering to support a different tabbing requirement. Do not use tab ordering to "fix" a bad page design. In the latter case, alter the order of the content in the markup itself instead of altering the order using `tabindex`.

When a component is included within another component, elements may have the same `tabindex` value. Place the included component in the page based on your preferred tab ordering among the elements in the parent page. WDK will ensure that `tabindex` collisions are resolved and elements are navigated in the order they appear in the character stream.

The following procedure describes how to add support for tab index to a WDK or custom control that does not have the `tabindex` attribute.

Supporting tab index in a custom control

1. Open the tag library descriptor for the custom control.
2. Add the `tabindex` attribute within the tag definition as follows:

```
<attribute>
  <name>tabindex</name>
  <required>>false</required>
  <rtexprvalue>>true</rtexprvalue>
</attribute>
```

3. Add `tabindex` attribute values on the JSP pages that contain the custom tag.
4. If the custom control provides its own rendering rather than that of a parent that supports tab index, call `renderTabIndex(StringBuffer)` API from the base `ControlTag` class in the rendering method.
5. Save the tag library descriptor and JSP files and restart the application server.

Setting initial focus on a control

1. Open the JSP page and locate the control that should have initial focus.
2. Add a focus attribute with a value of true. For example:

```
<dmf:text name='mytextcontrol' focus='true'...>
```
3. Save and close the file.



Caution: Tab index does not work well in a portal environment. Tab indexing is turned off in the app.xml <tab_ordering> element by default for portal environments.

Configuring action visibility and behavior

An action is visible if it can be performed. Invalid actions can be hidden by setting <display><hideinvalidactions> to true in app.xml. If this value is set to false, you can set visibility of invalid using the showifinvalid attribute on action controls.

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). Overridden by <hideinvalidactions> in app.xml.

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 57. Control visibility based on context

	Static	Dynamic
Visibility is calculated when:	Control is initialized	Associated actionmultiselect control in a datagrid is rendered.
Visibility is changed when:	Control is rendered	Single or multiple rows are selected in a datagrid
Argument tags are nested within:	Control	Associated actionmultiselectcheckbox control. (Use this even for generic and genericnoselect dynamic actions.)

Argument tags are passed to the action from the action control or from the associated actionmultiselect control. If actionmultiselectcheckbox controls are used, they can also contain arguments. Argument tags should not be nested within actionmenuitem. They will be ignored. For more information on passing arguments to action menu controls, refer to [Passing arguments to menu action items, page 229](#).

To enable an action menu item:

Use the dynamic attribute to enable and disable menu items based on the state of selected items on the page.

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 are selected. 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 whether items are selected.

1. In the **Edit** menu, set the **Delete** menu item dynamic attribute to "multiselect." The delete action will be executed on items on the page that are selected.
2. Set the **Rename** menu item dynamic attribute to "singleselect". This menu item will be enabled if one and only one object is selected.
3. Set the **View Clipboard** menu item dynamic attribute to "generic" so it will be enabled regardless of object selection.

These settings are shown below:

```
<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>
```

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 checkboxes are selected.

To combine dynamic control types

1. Create two controls, A and B.
2. Set the dynamic attribute of one control, control A, to singleselect.
3. Set the dynamic attribute of the other control, control B, to genericnoselect.
4. Set the showifdisabled and showifinvalid attributes to false on both of the controls.

5. Make sure that each control has a different action ID. The action definition used by control B can extend the action definition used by control A. For example, the action for control B can be defined as:

```
<action id="B" extends="A:custom/config/A_actions.xml"/>
```

For information on action controls and the dynamic attribute values, refer to [Action-enabled controls](#), page 121.

Validating user input

A validation control checks one input control for a specific type of error condition when a form is submitted and displays a message if an error is found. Control input can be validated by more than one validation control.

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.

Several validator controls are provided in WDK to validate data input. For more information on configuring individual validator controls, refer to *Web Development Kit and Webtop Reference*.

Validation errors do not prevent a control event from being fired. The component that uses validation controls must implement error handling for validation errors.

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. 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.

Data dictionary validation and value assistance are performed automatically for `docbaseattribute` and `docbaseattributelist` controls.

Configuring autocompletion

This topic describes how to configure autocompletion on an individual text control. For information on enabling or disabling autocompletion for the application, refer to [Enabling autocompletion, page 81](#). For information on adding autocompletion support to a control, refer to [Adding autocompletion support to a control, page 161](#).

Autocompletion for attribute controls is supported for drop-down list and text controls. Value assistance is read from the data dictionary, and the completion list for open lists is stored in the repository by attribute name for each user and is not specific to object type or repository.

You can disable autocompletion or change the autocomplete attributes for an individual control. The following attributes on a control that supports autocompletion can be configured:

- `autocompleteid`

Key that is used to store the autocomplete list in the user preferences. You can set the same ID on two text controls, for example, if they share the same set of suggestions. If the control has no ID specified, the ID will be generated based on the form id and text control name, in the form consisting of *formid_controlid*. For doctypeattribute controls that render a text control, an ID is generated using the attribute name, for example, keyword or title.

- `autocompleteenabled`

Boolean that enables or disables autocompletion on the control

- `maxauto completesuggestionsize`

Specifies the number of matching suggestions to display to the user. Default = 10

Tip: When the autocompletion list does not contain expected entries, make sure that the control has an `autocompleteid` with the same value as other controls that do show the expected entries.

There is no maximum character count for autocompletion entries. The length of the suggestion pop-up limited to twice the size of the text control on the JSP page.

Configuring context (right-click) menus

In datagrids that support row selection, a context menu for selected items is generated based on a menu that is defined in an action configuration file. The available actions are filtered by any presets that apply to the user context.

The global row selection flag in `app.xml` must be enabled for the application, and the datagrid JSP tag must have `rowselection` set to `true`. (If the attribute is not present, the default value is "true".) For information on enabling row selection for the application, refer to [Enabling datagrid features, page 81](#)

Context menus are defined in the `<menugroup>` element of an action configuration file. This element contains `<actionmenuitem>` elements that specify the actions that can be performed on the object type that is specified in the scope of the configuration file.

You can create submenus within the `<menugroup>` element by nesting a `<menu>` element with its own `<actionmenuitem>` elements. The following example from `dm_folder_actions.xml` defines a context menu for actions on selected `dm_folder` objects. Note that some actions support multiple selection and some support only single selection:

```
<menugroup id="context-menu">
  <actionmenuitem dynamic="multiselect" action="
    subscribe" .../>
  <actionmenuitem dynamic="multiselect" action="
    unsubscribe" .../>
  ...
  <menuseparator/>
  <actionmenuitem dynamic="singleselect" action="
    properties" .../>
  ...
  <menu nlsid="MSG_DETAILS">
    <actionmenuitem dynamic='multiselect' action='
      relationships' .../>
    <actionmenuitem dynamic='multiselect' action='
      locations' .../>
    <actionmenuitem dynamic='multiselect' action='
      showtopicaction' .../>
    ...
  </menu>
</menugroup>
```

When the user selects multiple objects, the common set of actions applicable to all selected objects is displayed.

Creating fixed menus

Actions can be grouped into menus so the set of actions available on a particular object type, user role, or other qualifier is specified in a single location. Menus can be defined in the JSP page using `<dmf:menu>` tags or in a menu configuration file using `<menu>` elements. The latter type of menu is preferable because you can extend it and reuse it. This topic describes configuration file menus only. Individual JSP page menus are supported for backward compatibility.

A menu is defined in a menu configuration file. This menu is included in a JSP page by its `id` attribute. In the JSP page, a `dmfx:menuconfig` tag references the menu, similar to the following:

```
<dmfx:menuconfig id='my_menu' />
```

The menu configuration file for this menu has an `id` that matches the `configid` attribute on the menu tag. In the following configuration file, two menus are defined: `my_menu` and `222_menuconfig`. The second menu is included as a submenu within the top-level menu:

```
<config>
  <scope>
    <menuconfig id='my_menu'>
      <menuitem name='aaa' label='Do A' />
      <menu id='111' name='111' label='B menu'>
        <menuitem name='b1' label='Do B1' onclick='event_handler' />
      </menu>
    </menuconfig>
  </scope>
</config>
```

```

        <actionmenuitem name='b2' label='Do B2' action='some_action' />
    </menu>
    <menuconfig id='222_menuconfig' />
</menuconfig>

<menuconfig id='222_menuconfig'>
    <menu id='222' name='222' label='C menu'>
        <menuitem name='ddd' label='Do C' />
    </menu>
</menuconfig>
</scope>
</config>

```

This example generates a menu with the following hierarchy:

Do A	B menu	C menu (included menu)
	Do B1	Do C
	Do B2	

The <menuconfig> element defines a menu that can be included within another <menuconfig> or referenced in a JSP page. In the example above, an empty <menuconfig> element is used to include the 'C menu' within the top-level 'my_menu'. The 'C menu' can also be used separately in another component because it is in a <menuconfig> element.

The elements in a menu configuration file, except the <menuconfig> element, generate JSP tags with the same name. Attributes on the configuration element are generated as attributes on the tag. For example, <menuitem name='file_help' nlsid='MSG_HELP' onclick='onClickHelp' runatclient='true' /> generates the JSP tag <dmf:menuitem name='file_help' nlsid='MSG_HELP' onclick='onClickHelp' runatclient='true' />. Just as for JSP tags, the nlsid key overrides a hard-coded label in the label attribute. NLS values are retrieved from the nls bundle of the component that contains the menu. If the menu item keys are not found in the bundle, then they are retrieved from the menu NLS bundle specified in custom/app.xml as the value of <menu>.nlsbundle>.

Table 58, page 136 describes the elements in a menu configuration file that can be used to generate a menu.

Table 58. Menu configuration elements

Element	Description
<menuconfig>	Defines a top-level menu. Contains at least one <menu> or <menuconfig> element. Has an id attribute that is used to include the menu into another menu or a JSP page.

Element	Description
<menu>	Defines a menu. If within a <menu> element, it defines a submenu. Can contain any combination of <menu> elements, which serve as submenus, <menuitem>, <actionmenuitem>, and <menuseparator> elements. Has the same attributes as the dmf:menu JSP tag.
<menuitem>	<menuitem> has the same attributes of the dmf:menuitem control. The name attribute is required. The menu item will not do anything unless you set a value for the onclick attribute.
<actionmenuitem>	<actionmenuitem> has the same attributes as the dmfx:actionmenuitem control. The action attribute is required and specifies the action that is launched by the menu item. The action must match the ID of an action definition in the application.
<menuseparator>	Generates a separator in the menu. Has the same attributes as dmf:menuseparator.

Menus can be extended and modified using the WDK extension mechanism. Refer to [Extending XML definitions, page 40](#) for information on how to extend a menu configuration. Refer to [Modifying configuration elements, page 33](#) for information on how to insert, remove, or override a menu item.

Tip: If you are reusing a menu in more than one component, put the menu into a <menuconfig> element. If you are making simple modifications to a WDK menu you will not be reused in other components, insert menu elements into the <menuconfig> element or its child elements using the modification mechanism. In the following example, the sample menu above is modified by inserting a menu item labeled **Do X** between **Do B1** and **Do B2**. For more information on modifying definitions, refer to [Modifying configuration elements, page 33](#).

```
<menuconfig modifies="my_menu:custom/config/mycomponent.xml">
  <insertafter path="menu[id=111].menuitem[name=b1]">
    <menuitem name="xxx" label="Do X" onclick="do_something"/>
  </insertafter>
</menuconfig>
```

To convert a JSP menu to an XML menu

1. Remove the tags contained within the dmf:menu tag on the JSP page.
2. Change the dmf:menu tag to dmfx:menuconfig tag with an id attribute value that references the id of the <menuconfig> element in the menu definition that you will create.
3. Create the menu definition in an XML file with the primary element <menuconfig>.

4. Replace each `dmf:menu` JSP tag with a `<menu>` element.
5. Within the `<menu>` element, replace each `dmfx:actionmenuitem` tag from the JSP menu with an `<actionmenuitem>` element. Replace each `dmf:menuitem` and `dmf:menuseparator` tag with `<menuitem>` and `<menuseparator>` elements as described in [Table 58, page 136](#).
Set the JSP tag attributes on the corresponding XML elements that replace them.
6. For each JSP menu item that is wrapped with a `dmf:clientenvpanel` tag, wrap the menu item element in the XML definition with a corresponding `<filter>` tag. The following example from the 5.3.x Webtop `menubar_body.jsp` page hides the New Process action menu item in the portal environment:

```
<dmf:menu name='file_new_menu' nlsid='MSG_NEW'>
...
  <dmfx:clientenvpanel environment='portal' reversevisible='true'>
    <dmfx:actionmenuitem dynamic='genericnoselect' name='
      file_newprocess' nlsid='MSG_NEW_PROCESS' action='
      newprocess' showifinvalid='true' />
  </dmfx:clientenvpanel>...
```

This menu item would be defined as follows:

```
<menu id='file' nlsid='MSG_NEW'>
...
  <filter clientenv='not portal'>
    <actionmenuitem dynamic='genericnoselect' name='
      file_newprocess' nlsid='MSG_NEW_PROCESS' action='
      newprocess' showifinvalid='true' />
  </filter> ...
</menu>
```

The `menubar` component allows you to specify all of the menus in the application menu bar. Extend the Webtop `menubar` component and specify the `menuconfig` IDs for each menu that should appear in the menu bar. For example, the Webtop `menubar` definition in `webtop/config/menubar_component.xml` inherits the following `menuconfig` IDs from the `webcomponent` `menubar` definition:

```
<menuconfigs>
  <id>menubar_file_menu</id>
  <id>menubar_edit_menu</id>
  <id>menubar_view_menu</id>
  <id>menubar_tools_menu</id>
  <filter entitlement="recordsmanager">
    <id>menubar_rpm_menu</id>
  </filter>
</menuconfigs>
```

The following example in a custom `menubar` component definition modifies the menu to add a menu item at the end:

```
<component id="menubar" modifies="menubar:webtop/config/menubar_component.xml">
  <insert path='menuconfigs'>
    <id>menubar_mymenu</id>
  </menuconfigs>
</component>
```

Filtering by object type

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 the type dmc_jar. Any custom type can be substituted to create a filter.

Note: If the custom type does not exist in the repository, the drop-down 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.

If you create a custom filter, add it to every component that can display your custom object type. The following example adds the custom filter to the Webtop objectlist component.

To create a custom filter

1. Create a new configuration file in custom/config, for example, objectlist_modifications.xml.

2. Open the file and add the required XML structure:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<config>
<scope>
</scope></config>
```

3. Between the scope elements, add the primary element definition that you are modifying:

```
<component modifies="homecabinet_list:
webcomponent/config/navigation/homecabinet/homecabinet_list_component.xml">
```

4. Add your objectfilter element to the objectfilters element. Substitute the custom object type for the value of the <type> element:

```
<insert path="objectfilters">
  <objectfilter>
    <label>Show Jars</label>
    <showfolders>>false</showfolders>
    <type>dmc_jar</type>
  </objectfilter>
</insert></component>
```

5. Refresh the cached configurations by navigating to wdk/refresh.jsp, and then view your home cabinet. This example assumes that you have added a file of type dmc_jar to your home cabinet.

[Figure 7, page 140](#) displays the results of the files filter, which resolves to dm_document and its subtypes and excludes folders.

Figure 7. Home cabinet with standard files filter

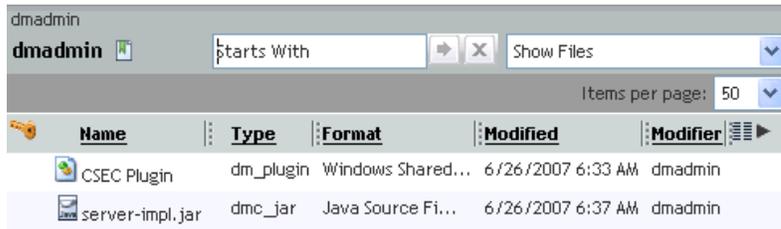
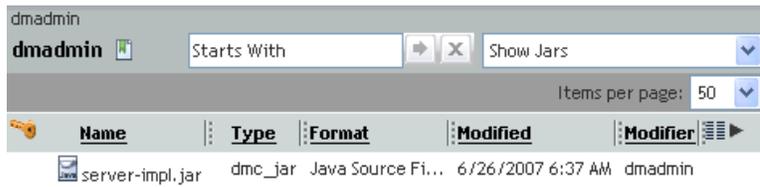


Figure 8, page 140 displays the results of a custom filter.

Figure 8. Home cabinet with custom filter



Configuring rich text

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 invokes 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.

The datatype 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 by the 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. It is installed as a DLL if enabled in app.xml. (This is a different editor from the xforms rich text editor.)

The following elements can be configured.

Table 59. Rich text configuration elements

Element	Description
<inputfilter>	Processes images and links that are entered by the user. Remove the comments from this element to use the RichTextInputFilter class.

Element	Description
<outputfilter>	Processes image URLs for display in IE and Mozilla. Remove the comments from this element to use the RichTextOutputFilter class.
<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 60. Rich text editor configuration elements (<editor>)

Element	Description
<ieminversion>	Specifies the minimum version of IE supported by editor
<mozillaminversion>	Specifies the minimum version of Mozilla supported by editor
<netscapeminversion>	Specifies the minimum version of Netscape supported by editor
<safariminversion>	Specifies the minimum version of Safari supported by editor
<iemaxversion>	Specifies the maximum version of IE supported by editor

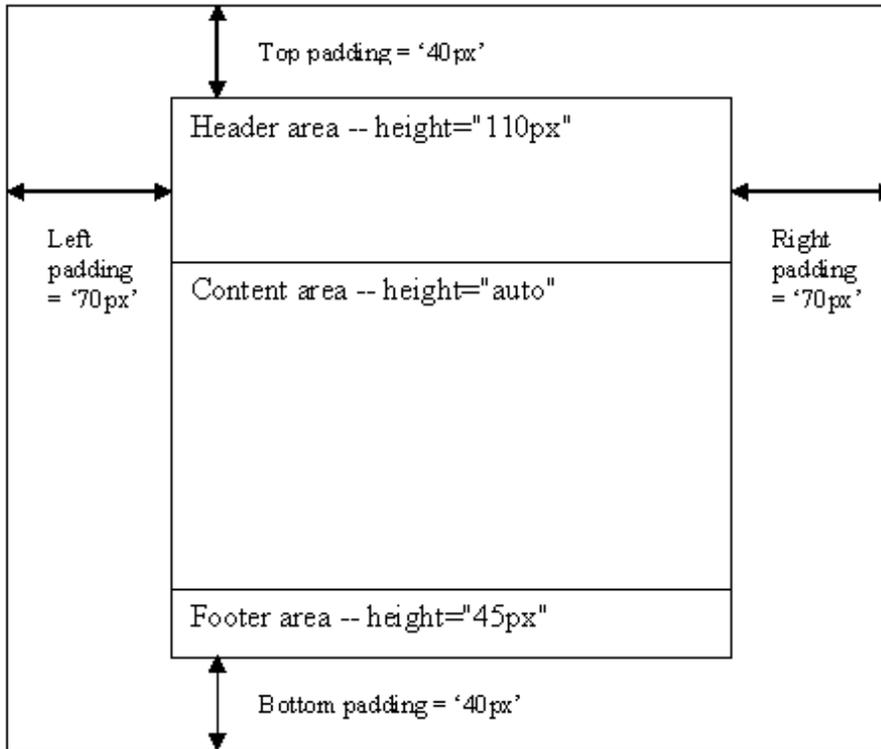
Element	Description
<mozillamaxversion>	Specifies the maximum version of Mozilla supported by editor
<netscapemaxversion>	Specifies the maximum version of Netscape supported by editor
<safarimaxversion>	Specifies the maximum version of Safari supported by editor

Configuring scrollable panes

If your display is likely to be larger than the browser window, 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 labeled parts is configurable as an attribute on the paneset or pane control:

Figure 9. Scrollable pane controls

For more information on configuring each control, refer to the reference information on the control in *Web Development Kit and Webtop Reference*.

Configuring JSP fragments

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 Application Connectors 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 by the branding service, page 363](#)). In the example, 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..

Adding a tooltip

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 controls expose tooltip as a configurable attribute on the JSP tag, such as : link, text, label, button, option, tab, and many others. Other controls 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 enabling this accessibility feature, refer to [Enabling accessibility \(Section 508\), page 70](#).

Adding images and icons

You can specify the path to individual images or icons in an image control 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.

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.

All graphics in the /images and /icons directories under a /theme directory 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 [Providing image descriptions, page 453](#).

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 new theme, page 356](#).

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 359](#).

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. (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' />
```

Hiding a control

You can configure a control on the JSP to hide it, rather than removing it from the JSP page. You may want to do this because your component needs information that is set by the control but you do not want to expose it to users. For example,

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>
```

Control customization overview

The following topics provide information required to customize controls.

Control classes

Every control class implements member variables, server-side getter and setter methods, and event handlers for a control. A control class has a corresponding control tag class that 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 rendered UI.

The Control class in the documentum.web.form package is an abstract class. The Control class exposes type-safe methods to access contained controls and the parent container. Control instances on the server are bound to successive requests, so member variable values are preserved across requests.

The following table describes properties that are common to all controls:

Table 61. Control class properties

Property	Description
name	String that identifies the control so it can be manipulated by server-side event handlers
nlsid	National Language Support (NLS) ID, which is used by Form.getString() to look up a localized string. The string is displayed as a UI element.
datafield	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 datagrid.
elementName	Name of a control when it is rendered as an HTML element
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 62. 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. For example: Button, Image, Link, Label.

Class	Use
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 does not need access to tags contained within the control. The <i>value</i> attribute accepts user input. The corresponding control class must extend StringInputControl. For example: Text, Password, Hidden.
BooleanInputControlTag	Extends ControlTag. Extend this class when your control accepts a Boolean input from the user but does not need access to tags contained within the control. The <i>value</i> attribute accepts user input. The corresponding control 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 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 Java EE 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.

How controls and tags work together

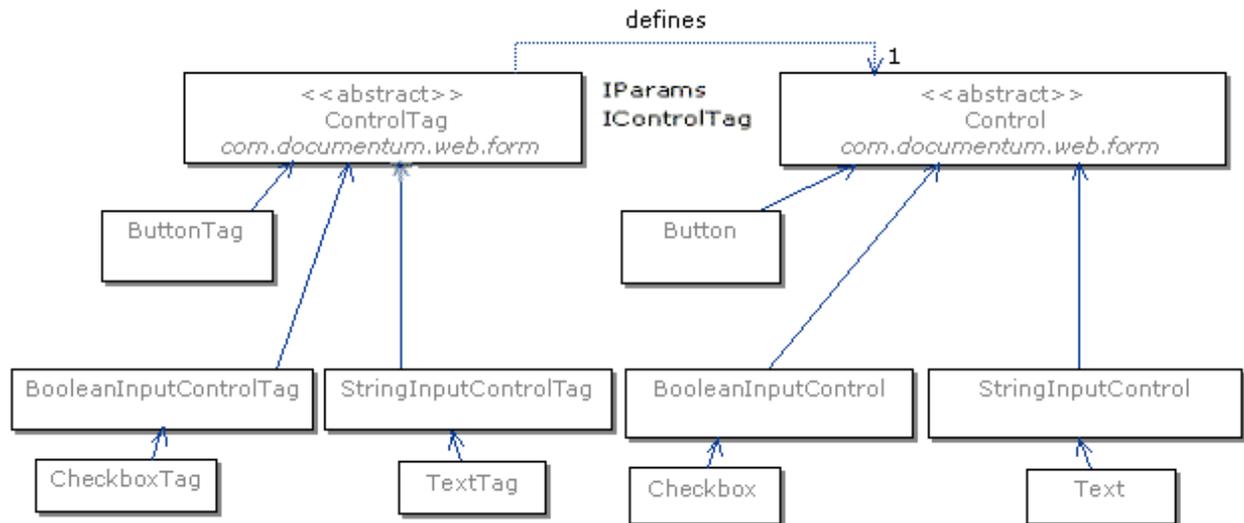
Controls and tags interact as follows:

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 10. Control and ControlTag relationship



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 172](#))

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 121](#) 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 63. Event attributes

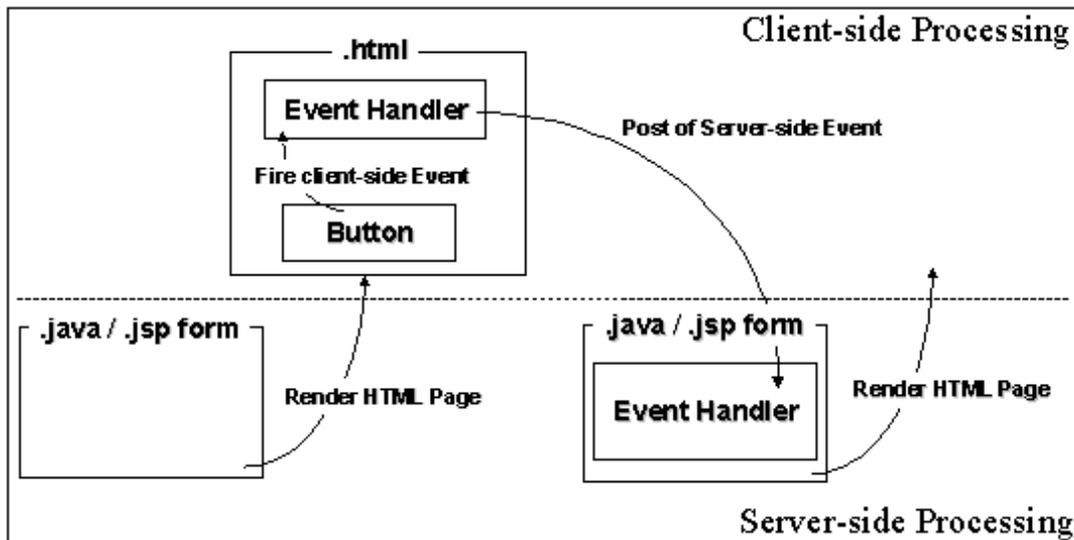
Attribute	Description
<code>onchange</code>	Sets the event that is fired when the control is changed by the user. The <code>onchange</code> event handler cannot run on the client (<code>runatclient</code> cannot be true). The <code>onchange</code> event is not handled immediately. It is handled on the server only when the form is submitted (for example, by an <code>onclick</code> event). Some controls do not implement an <code>onchange</code> event.
<code>onselect</code>	Sets the event that is fired when the user selects the control, such as an option in a list. The <code>onselect</code> event is handled immediately, either on the server (when <code>runatclient=false</code>) or on the client (when <code>runatclient=true</code>). Some controls do not implement an <code>onselect</code> event.
<code>runatclient</code>	Specifies that the <code>onchange</code> event should run on the client, not the server. Some controls do not support a <code>runatclient</code> attribute.

Attribute	Description
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.
defaulttonenter	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 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 defaulttonenter=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 defaulttonenter control.

Refer to the individual control descriptions in *Web Development Kit and Webtop Reference* for the specific events that are defined for each control.

Control events are raised when a form URL requests operations on the server. The form (URL) is posted again as a recall operation. The following diagram illustrates the interaction between client-side and server-side processing:

Figure 11. 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 Java EE 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='/wdk525spl/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");
}
```

Choosing server-side or client-side event handling

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.

Choosing a control superclass

The following table can help you select a superclass for your control:

Table 64. 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 checkbox)	Use BooleanInputControl and BooleanInputControlTag. Override renderStart() and renderEnd() to write the HTML rendition.

Customizing controls

This topic addresses common control customization tasks.

Getting controls programmatically

The WDK framework gives each control 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.

You can get a control using `Form.getControl()`, passing in the name of the control and the control class name. If the requested control does not exist, or if you do not pass in the control class argument, a new control instance is created. To get a control and read its value, you must get the control after the component has rendered all controls.



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.

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" .../>
```

Example 3-1. Retrieving a control value

You can retrieve the value of control attributes by using accessor methods for the controls. You must retrieve values after the component has rendered all controls. The best way to do this is either in the `onRenderEnd` event handler or in some other form event handler.

The following example gets the value of a text control `dqLeditor.jsp` that sets column width. The JSP tag:

```
<dmf:text name='<%=DQLEditor.CONTROL_COLUMNWIDTH%>' size='4'
  onchange='onSetColumnWidth' />
```

The component class `DQLEditor` retrieves the control value by calling `getValue` on the control:

```
public void onSetColumnWidth(Text text)
{
    m_iColumnWidth = Integer.parseInt(text.getValue());
    ...
    text.setValue(new Integer(m_iColumnWidth).toString());
    ...
}
```

To get a control value in a method that is not a control event handler, first get the control by name, passing in the control class. The following example from the `RoomHomePage` class in `Webtop` gets the control and then gets its value:

```
Radio subscribe = (Radio) roomNavigation.getControl(
    "subscribe", Radio.class);
...
if (subscribe.getValue())
{...}
```

Naming a control — Named controls are retained in server memory and bound to each HTTP request, so 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 to allow access to the control state in your component.

Example 3-2. Accessing an unnamed 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());
    ...
}

```

Accessing contained controls

To access a control that is contained within another control, loop through all the controls of the container. The following example gets all the controls in the component JSP page to find a panel control, then sets the panel to invisible:

```

private void setPanelsInvisible()
{
    Iterator iterator = getContainedControls();
    while(iterator.hasNext())
    {
        Control control = (Control)iterator.next();
        if(control.getTypeName().equals("Panel"))
        {
            ((Panel)control).setVisible(false);
        }
    }
}

```

Accessing controls by JavaScript

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.

The WDK framework gives each control 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. The ID is not used by the WDK framework, but you can use it in JavaScript. JavaScript does not 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)`.

If you do not give the control an ID using the `id` attribute, 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 3-3. 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' defaultonenter='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>
```

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 3-4. 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.

All validator controls extend the Label control and implement the `IValidator` interface. This interface defines three methods: `validate()`, `getIsValid()`, and `getErrorMessage()`.

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.

The `BaseValidator` 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

Creating a validator control

This topic describes how to create a custom validator, with a working example. The following example illustrates the steps in building a simple password validator that rejects passwords 1111 or 1234.

To create a validator control

1. Develop a validator tag.

Add a custom custom tag library definition file (*.tld) to `WEB-INF/tlds`, with an entry for your custom validator, for example:

```
...
<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>

```

2. Create the validator tag class.

a. Extend BaseValidatorTag, for example:

```

package com.mycompany;
import com.documentum.web.form.control.validator.BaseValidatorTag;

public class PasswordValidatorTag extends BaseValidatorTag
{
}

```

b. Add accessor methods for any custom validator attributes, for example:

```

protected Class getControlClass()
{
    return PasswordValidator.class;
}

```

c. Override doValidate() to provide custom validation:

```

protected boolean doValidateValue(String strValue)
{
    boolean bValid = true;
    if (strValue.equals("1111") || strValue.equals("1234"))
    {
        bValid = false;
    }
    return bValid;
}

```

3. Use the validator in a component JSP page.

a. Create a changepassword definition that modifies the changepassword component in /wck/config.

b. Add your modification as follows:

```

<component id="changepassword" modifies="
  changepassword:wck/config/changepassword_component.xml">
  <replace path="pages.filter[clientenv=webbrowser]">
    <start>/custom/changepassword.jsp</start>
  </replace></component>

```

c. Edit your custom changepassword.jsp page to use your custom tag, for example:

```

<%@ page contentType="text/html; charset=UTF-8" %>
<%@ page errorPage="/wck/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>
...
<tr>
  <td>New Password:</td>
  <td><dmf:text name="newpassword" size="40"/>&nbsp;  
    <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>
...

```

Implementing multiple selection

Web applications can support actions on more than one selected object. Objects are selected by single-click on a row, shift-click for contiguous row, control-click for multiple objects, or right-click for a context menu. The `actionmultiselect` control provides support for invoking actions on multiple selected items. To display checkboxes for selection, add `actionmultiselectcheckbox` controls and enable checkboxes in your custom `app.xml` file by setting the value of `<desktopui>.<datagrid>.<rich-ui>` to `false`.

Note: When rich UI is enabled, checkboxes are not rendered. 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 table with HTML table row controls (`<tr>`), typically within a `datagrid`. To pass arguments to all actions, add argument controls to the `actionmultiselect` control. Within each WDK control in a table cell, you can embed argument controls for the associated dynamic actions. Argument controls can also define context values that will be used to return 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: Use only one `actionmultiselect` control per set of open frames in your application. If you have more than one `<dmfx: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. You can configure the application to test action preconditions only when they are executed instead of on page rendering. Set the `onexecutiononly` attribute of the precondition element to `true` as follows:

```
<precondition onexecutiononly="true" class=.../>
```

Caching component arguments for multiple selection — If your action can be called for multiple objects, you can direct the container class to cache the component arguments rather than pass them on the URL, possibly enhancing performance. To do this, modify the action definition that calls your container and set the `<container>` attribute `storeargsinmemory` to `true`. For example:

```
<action id="myaction">
...

```

```
<execution class="com.documentum.web.formext.action.LaunchComponent">
  <component>multiobjectsaction</component>
  <container storeargsinmemory="true">multiobjectscontainer</container>
</execution>
</action>
```

Getting multiple selection arguments in the component — There are two ways to get the arguments for multiple selections in your component class:

- Use `MultiArgumentContainer` for your component

For example, the `sendtodistributionlist` component uses this container. The `MultiArgumentContainer` 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 < strComponentArgs.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");
```

Customizing drop-down lists

The `DataDropDownList` control has a `query` attribute that can be used to provide options.

Example 3-5. Overriding a list of options

To set the options dynamically in your component class based on some query or test, get the control and set options 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);
}

```

Note: You cannot replace options in a datadropdownlist control because those options are provided by a datafield. You can, however, change the control type in the JSP page to dropdownlist and provide the options in your component class.

Adding autocompletion support to a control

If your control extends the WDK text or dropdownlist controls, enable autocompletion by adding the following attributes to the tag library descriptor entries for the control:

```

<attribute>
    <name>autocompleteenabled</name>
    <required>>false</required>
    <rtexprvalue>>true</rtexprvalue>
</attribute>
<attribute>
    <name>autocompleteid</name>
    <required>>false</required>
    <rtexprvalue>>true</rtexprvalue>
</attribute>
<attribute>
    <name>maxautoCompletesuggestionsize</name>
    <required>>false</required>
    <rtexprvalue>>true</rtexprvalue>
</attribute>

```

For information on these attributes, refer to [Configuring autocompletion, page 134](#)

To implement autocompletion in other controls, implement `IAutoCompleteEnabledControl` in the control class. The method `getInputValue()` should get the value that the user has entered and add it to the autocompletion list. For example, the `Text` class implements the method in the following way:

```

public String getInputValue()
{
    return getValue();
}

```

The tag class must bind the autocomplete object during rendering. Call `renderAutoCompleteTextBinding()`, which is implemented in the `ControlTag` class, as in the following example from the `TextTag` class:

```

protected void renderEnd(JspWriter out)
    throws IOException
{
    //...
    if (text.isAutoCompleteEnabled())
    {

```

```
// generate the JavaScript to support AutoComplete
// generate the auto complete list
renderAutoCompleteTextBinding(buf, Text.EVENT_ONVALUECHANGE, null);
}
```

Add the autocompletion attributes to the tag library descriptor entries for the custom tag as described at the beginning of this topic.

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. You can modify which control is used to render an attribute by specifying a tag class that extends `DocbaseAttributeValueTag`. These classes will then be used to display or handle data in a `docbaseattribute`, `docbaseattributevalue`, or `docbaseattributelist` control.

These custom formatters, handlers, and tag classes are registered in a `docbaseobject` configuration file whose root element is `<docbaseobjectconfiguration>`. To use these classes, reference the configuration in a JSP page 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 a `docbaseobject` tag, the default configuration is used. This default configuration has an `id` of "attributes" and is located in `webcomponent/config/library/docbaseobjectconfiguration_dm_sysobject.xml`.

The following topics describe the configuration file, formatters, value handlers, and `docbaseattributelist` lookup process.

Configuring a required attribute

If an attribute is not specified as required in the data dictionary, you can configure it to be required in a `docbaseobjectconfiguration` file by using the `valuetagclass` `RequiredDocbaseAttributeValue`. The following example can be added to in a custom configuration file, located in `custom/config`, to make the `object_name` attribute required when it is rendered in the **Create** or **Import** components' UI:

```
<docbaseobjectconfiguration modifies='
attributes:webcomponent/config/library/docbaseobjectconfiguration_dm_sysobject.xml'>
<insert path='names'>
  <attribute name='object_name'>
    <valuetagclass>
      com.documentum.web.formext.control.docbase.RequiredDocbaseAttributeValueTag
    </valuetagclass>
  </attribute>
</insert>
</docbaseobjectconfiguration>
```

Creating an attribute formatter

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;
}
```

Creating an attribute value handler

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);
}
```

Creating a custom attribute tag class

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. 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>
  ...
</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 example 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;
```

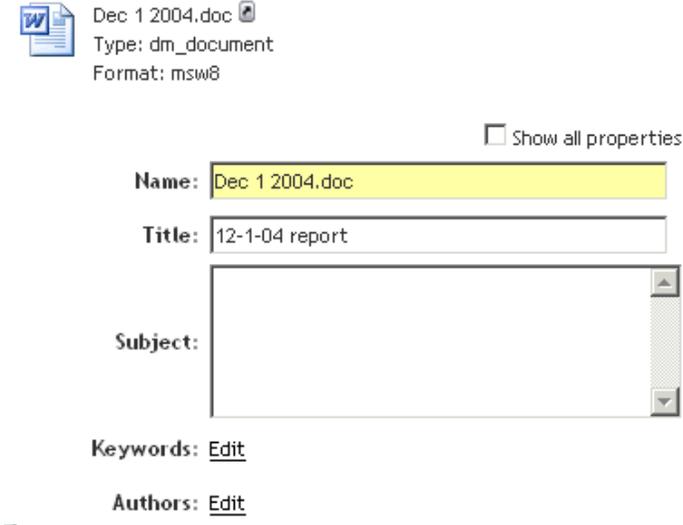
```
import javax.servlet.jsp.JspWriter;

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 13. String attribute rendered as TextArea control

Adding a custom element to be used in attribute handling

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());
}
```

Adding an attribute editor

You can specify a component that will be launched to edit a specific attribute (`<attribute name=...>`) or attributes of a specific type (`<attribute type=...>`). For example, the default `docbaseobjectconfiguration` definition specifies that the `versionlabels` component should be used to edit the `r_version_label` attribute.

Parts of the 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 ID-type attributes. This formatter 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.

Tip: Some attributes, such as r_version_label, are read-only in the data dictionary but are editable in WDK-based applications. To make these attributes read-only, remove the <valuehandler> element for them in the docbaseobjectconfiguration file.

Default attribute handling

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 65. Default attribute handling

Attribute	Customization
a_storage_type	Renders storage name instead of integer
r_object_id	Renders object ID as text, overrides the rendering of the ID type

Attribute	Customization
i_chronicle_id	Renders chronicle ID as text, overrides the rendering of the ID type
r_policy_id	Renders policy name instead of ID
r_version_label	Sets changes to version labels, uses versionlabels component instead of docbaserepeatingattributes component
r_current_state	Renders state name instead of state number
r_resume_state	Renders state name instead of state number
type = id	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.
type=rich_text	Specifies handlers for attributes of the richtext pseudoattribute type. This customization is applied to attributes in a scoped attributelist definition that contains in <pseudo_attributes> element. The attribute within this element must have the type rich_text, for example: <attribute name=folder_description type=rich_text ...>

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 3-6. 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());
}
```

Generating control 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.

Example 3-7. 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 3-8. 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(objectId);
        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();
}

```

Example 3-9. Rendering an HTML link

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 the link will appear active. The actual action is done by the onclick event handler that is specified in the HTML tag.

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.

Getting and displaying the 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.

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.

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 66. Folder path display rules

Argument value	Return value
<code>"/a/b/c/d/e"</code>	<code>"/a/b/c/d/e"</code>
<code>"/a/b/c/d/e"</code>	<code>"/.../c/d/e"</code>
<code>"/a/b/c/d/e"</code>	<code>"/.../d/e"</code>
<code>"/a/b/c/d/e"</code>	<code>" "</code>
<code>" "</code> or null	<code>" "</code>

Programming control events

The following topics describe control events: How to fire them on the client and server and how to handle them on the client and server.

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 a attribute on the control in the JSP page. For example, the following button control in `dqlEditor.jsp` specifies an `onclick` event handler that corresponds to a handler method `onClickExecute()` in the component class:

```
<td colspan='10'><dmf:button nlsid='MSG_EXECUTE'
  onclick="onClickExecute" tooltipnlsid='MSG_EXECUTE' /></td>
```

Controls that have an `onclick`, `onselect`, or `onfileselect` event submit the form and are handled immediately. Controls that have an `onchange` event do not submit the form and are handled only

after the form has been submitted. Controls that have an onload and onunload event are handled when the browser loads or unloads the page.

A control can also fire a client event to be handled on the client by JavaScript. Controls that can fire a client event have a `runatclient` attribute, and you must set this attribute to true. You can register an event handler for the client event that is raised in another active component. For information on registering client event handlers, refer to [Registering a client event handler, page 179](#).

The following topics describe different ways of firing and handling control events.

Handling a control event in the component class

A server-side event handler must be in the class of the component that owns the JSP page on which the control is found, in a parent class of the component, or in the class of the container that contains the component. In the following example from `setRetentionDate.jsp`, when the user clicks the **Retention Period** radio, the `onclick` event specifies an `onRetentionSelection` event handler, as follows:

```
<dmf:radio ... nlsid='MSG_RETENTION_PERIOD' ...
  onclick='onRetentionSelection' />
```

This handler is in the component class `SetRetentionDate` as follows:

```
public void onRetentionSelection(Radio radioControl, ArgumentList args)
{
    if(radioControl.getName().equals(RADIO_RETENTION_PERIOD_CONTROL))
    {
        disableRetentionPeriodControls(false);
    }
    else
    {
        disableRetentionPeriodControls(true);
    }
}
```

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)
  {
    ...
  }
%>
```

For information on passing arguments from the control to the component event handler, refer to [Passing arguments from a control to a component, page 283](#).

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`.

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* is the ID of the form to submit. If null, the first form on the page is assumed.
- *strSrcCtrl* is the ID of control that fires the event (optional).
- *strHandlerCtrl* is the ID of the control that handles the event (optional).
- *strHandlerMethod* is the Java method name of the event handler in a class on the Java EE application server.
- *strEventArgsName* is the event argument name.
- *strEventArgsValue* is the event argument value.

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. In the following example from `launchRepositorySelector.jsp`, the body `onload` event calls the JavaScript function `invokeLaunch` and posts the server event `onLaunch`, which is then handled in the component class `LaunchRepositorySelector`:

```
<script>
function <%=JavascriptUtil.namespaceScriptlet(request, "invokeLaunch()")%>
{
    <dmf:postserverevent handlermethod="onLaunch"/>
}
</script>
```

In `clipboard.jsp`, the JavaScript function `onRemoveItems()` is called by a button event:

```
<dmf:button onclick='onRemoveItems' runatclient='true'.../>
```

The server event is posted with arguments for the selected items:

```
function onRemoveItems()
{
  var items = getSelectedItemIds();
  if (items.length > 0)
  {
    <dmf:postserverevent formid='<%=form.getElementName()%>'
      handlermethod='onRemove' eventargname='itemIds' eventargvalue='items' />
  }
  else
  {
    alert('<%=form.getString("MSG_NOITEMSSELECTED")%>');
  }
}
```

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 `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 Java EE 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 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.

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 a client event handler, page 179](#) 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 client event handler is contained within the same frame's JSP page or an included JavaScript file:

```
function onRefreshFrames(strExclude)
{
    for (var iFrame=0; iFrame < window.frames.length; iFrame++)
    {
        window.frames[iFrame].location.reload();
    }
}
```

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 Java EE server, use the `<dmf:frameset>` and `<dmf:frame>` tags from the WDK `dmf:form` library to generate framesets.

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:

```
<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>

```

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

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 onClassicView(view)
{
    changeFrameLocationInFrameset(parent.parent, "view", "
    <%=request.getContextPath()%>/webtop/classic/classic.jsp");
    fireClientEvent("SetView", view);
}

```

Handling a client-side control event

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>

```

You can add custom arguments that are passed to the client-side event handler method by using the `<argument>` tag within the control 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>

```

Note: Control tags that have a <bodycontent> value of "empty" in the tag library descriptor do not support argument tags.

Client-side <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 navigation from a client-side function

Use client-side navigation functions 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 value.

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

```

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

```

You can also issue a URL to a WDK component directly within a JavaScript event handler, similar to the following:

```

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 reference by `<%@ include file='doclist_thumbnail_body.jsp' %>` is in the same directory as the including file. The included file reference by `<%@ include file='/wdk/container/modalDatagridContainerStart.jsp' %>` is in the wdk subdirectory of the web application.

Registering a client event handler

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);

```

`strSrcWindowName` is the optional source frame or window name. `strEventName` is the event name. `fnEventHandler` is 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, using JavaScript syntax, and registered, using WDK syntax.

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: `WebformScripts.properties` located in `WEB-INF/classes/com/documentum/web/form`. Each registry entry has the following syntax:

```
index_name.property=value
```

where:

- *index* is a number indicating the order of inclusion in the HTML form. *name* is the logical
- *name* is the logical name of the script
- *property* is the type of property. Valid values are `href`, `language`, and `trace`.
- *value* is the 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` forces inclusion of the `href`. Set to `true` for a portal environment.

WDK registers its own scripts to support the client-side infrastructure. These scripts are included in any form that contains a `dmf:webform` tag. Do not modify the script registry entries for the WDK scripts.

Table 67. WDK scripts

Script file	Description
<code>trace.js</code>	Enables client-side tracing
<code>wdk.js</code>	
<code>locate.js</code>	Locates browser window frames
<code>events.js</code>	Stores and retrieves the scroll position
<code>scroll.js</code>	Enables client-side events
<code>formnavigation.js</code> and <code>componentnavigation.js</code>	Enable navigation between forms and components
<code>framenavigation.js</code>	Enables frame and browser identification by assigning IDs to frames and browser windows, and sets an in-memory cookie to identify the browser window
<code>help.js</code>	Enables the online help to launch in a separate browser window
<code>modal.js</code>	Enables modal dialogs
<code>contenttransfer.js</code>	Registers an event handler for HTTP download
<code>tree.js</code>	Supports tree node expansion and collapse in client JavaScript functions
<code>appintgevents.js</code>	Supports Application Connectors client events

Script file	Description
windows.js	Provides JavaScript window functions
controls.js	Supports button client-side display states
autoComplete.js	Displays autocompletion container

Script tracing — If script tracing is enabled (refer to [Using client-side tracing, page 447](#)), 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");
}
```

Firing events between frames

Client-side processing is sometimes 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.

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 a client event handler, page 179](#).)

WDK JSP pages can fire an inter-frame event using the `fireClientEvent()` function. This function is defined in the JavaScript file `events.js`, which is automatically included in all rendered HTML pages. 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 provided to `fireClientEvent` are passed automatically to the registered event handler.

Example 3-10. Firing an inter-frame event

In the following example from Webtop objectlist.jsp, the onClickContentObject event is fired when the user clicks an object in the object list:

```
function onClickObject(obj, id, type, isFolder)
{
    ...
    fireClientEvent("onClickContentObject", id, type);
    ...
}
```

The parent frame JSP page, classic.jsp, registers an event handler for this event:

```
registerClientEventHandler(null, "onClickContentObject", onClickContentObject);
```

The handler function onClickContentObject posts a server event to refresh the browser frame, which is defined further on the page:

```
function onClickContentObject(id, data)
{
    frames["browser"].safeCall("postServerEvent", null, null, null, "
    refreshTreeFromId", "componentId", g_lastComponentId, "objectId", id, "
    data", data );
}
...
<dmf:frame nlsid="MSG_BROWSER" name='browser' .../>
```

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 3-11. 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('>');
```


Configuring and Customizing Data Access

The following topics describe common tasks in configuring and customizing data access:

- Configuring data access
 - Databound controls, page 186
 - Configuring data display and selection, page 186
 - Configuring drop-down lists, page 194
 - Configuring attribute display, page 195
 - Generating lists of attributes for display, page 196
 - Configuring a "Starts with" filter, page 201
- Customizing data access
 - Providing data to databound controls, page 202
 - Accessing datagrid controls, page 203
 - Getting data in a component class, page 204
 - Getting data from a query, page 205
 - Getting or overriding data in a JSP page, page 206
 - Refreshing data, page 207
 - Caching data, page 207
 - Rendering data with result sets, page 208
 - Adding custom attributes to a datagrid, page 210
 - Implementing non-data dictionary value assistance, page 213

Refer to [Chapter 3, Configuring and Customizing Controls](#) for information on general control configuration and customization.

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.

All of the standard tag library controls can bind to data from a data provider. 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 overrides the existing label or value attribute. In the following example from `objectgrid_classic.jsp`, the `datafield` provides the label for policy state, overriding the label control's label attribute,:

```
<dmf:label datafield='r_current_state' />
```

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' />
```

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 does not have a repository session. Refer to [Providing data to databound controls, page 202](#) for information on establishing a session and data provider.

Configuring data display and selection

The following topics describe configuration of controls that are bound to data in a repository table or other data source.

Configuring column headers and resizing

To display resizable columns or fixed column headers that remain in place as the user scrolls down, you must enable these in the `datagrid`. Additionally, this `datagrid` functionality must be enabled in `app.xml`: the value of `<desktopui>.<datagrid>.<rich-ui>` must be set to `true`. Refer to [Enabling datagrid features, page 81](#) for more information.

Resizable columns — Some columns should be resizable. Columns that contain fixed-width content such as icons do not need to be resizable. To enable this feature, set the `resizable` attribute on a `datagridTh` tag to `true`.

Supporting column resize in custom components — For listings that extend the WDK doclist component or the Webtop objectlist component, resizing is supported. For custom listing classes, implement support to persist the column width. Add the following method in your component implementation:

```
public void onInit(ArgumentList args)
{
    ...
    initColumnWidths();
}

protected void initColumnWidths()
{
    //replace preference_id_string with string to persist the preference
    //replace CONTROL_GRID with datagrid ID
    m_widthsHelper = new DatagridColumnWidthPreferenceHelper(
        this, preference_id_string, CONTROL_GRID;
    }
private DatagridColumnWidthPreferenceHelper m_widthsHelper;
```

For components that extend ObjectGrid, you must perform the following steps to support column resize:

1. Replace <th> and <td> tags with <dmf:datagridTh> and <dmf:datagridRowTd> tags and set resizable to true on all <dmf:datagridTh>.
2. If the datagrid is not named <%= ObjectGrid.GRID_NAME %>, then specify the name by overriding getDataGridName. For example, if your datagrid name in the JSP page is <dmf:datagrid name='myGrid'>, you override getDataGridName as follows:

```
protected String getDataGridName()
{ return "myGrid"; }
```

3. Override getColumnPreferenceId so that the preference can be saved. For example, the relationships component class defines the preference in the following way. If you wish to provide a <preference> element in the display_preferences component definition, you can configure default columns.

```
protected String getColumnPreferenceId ()
{
    String prefId = lookupString("columnpreferenceid", false);
    return (prefId==null||prefId.length()==0)?PREF_CLASSIC_RELATIONSHIPS: prefId;
}
private static final String PREF_CLASSIC_RELATIONSHIPS = "
    application.display.classic_relationships_columns";
```

Note: If rowselection is set to false on the datagrid or in app.xml, this feature is not available.

Fixed column headers — The datagrid control can render fixed column headers, which are not affected by vertical scrolling of the data rows inside the grid. The generation of fixed headers is controlled by the fixedheaders boolean attribute of the datagrid control tag. It is false by default and must be enabled by the page author. The datagrid must contain datagridTh and datagridRowTd control tags, which ensure that the datagrid HTML table markup has the proper column alignment.

The `datagridTh` tag is used in place of `<th>` elements in the grid header row. The `datagridRowTd` tag is used in place of `<td>` elements in the grid body. The outline of `datagrid` markup for fixed column headers is similar to the following

```
<dmf:datagrid ...rowselection="true" fixedheaders="true">
  <tr>
    <dmf:datagridTh scope='col'>
      <dmf:datasortlink name='sortcoll' column='object_name' .../>
    </dmf:datagridTh>
    <dmf:datagridTh scope='col'>
      <dmf:image name='prop' nlsid='MSG_PROPERTIES' .../>
    </dmf:datagridTh>
  </tr>
  <dmf:datagridRow>
    <dmf:datagridRowTd scope='row'>
      <dmf:label datafield='object_name' />
    </dmf:datagridRowTd>
    <dmf:datagridRowTd>
      <dmfx:actionimage nlsid='MSG_PROPERTIES' action='properties'...>
      <dmfx:argument name='objectId' datafield='r_object_id' />
      <dmf:argument name='type' value='dm_acl' />
    </dmfx:actionimage>
    </dmf:datagridRowTd>
  </dmf:datagridRow>
</dmf:datagrid>
```

Note: If `rowselection` is set to `false` on the `datagrid` or in `app.xml`, this feature is not available.

The height of the `datagrid` can be configured in two ways:

- auto height (no height value specified)

The grid takes up the remaining space in the frame. Tags on the page past the `datagrid` will not be displayed unless they are placed between the closing `datagridRow` tag (`</dmf:datagridRow>`) and the closing `datagrid` tag (`</datagrid>`). Frame scrolling is disabled, and only `datagrid` scrolling is enabled.

- fixed height

Additional tags past the `datagrid` should be placed within the footer (anywhere between the end of `datagridRow` tag and the end of `datagrid` tag). Frame and `datagrid` scrolling are enabled.

If you provide a value for the `height` attribute on a `datagrid` control, scroll bars will be generated for the contents of the `datagrid` that extend beyond the value specified in the `height`.

Configuring datagrid row selection

Datagrids are enabled for mouse or keyboard row selection and right-click context menus. Refer to [Enabling datagrid features, page 81](#) for information on how to enable or disable this feature for the entire application. Right-click context menus can be added for the selected rows. Refer to [Configuring](#)

[context \(right-click\) menus, page 134](#) for information on how to create or change a right-click context menu for selected objects in a datagrid.

When row selection is enabled in app.xml, checkbox tags and actionmultiselect checkbox tags within a datagrid on the JSP page hidden. (They are rendered with a CSS rule of display:none.) Rows can then be selected by mouse events or keyboard combinations. To disable row selection for an individual datagrid row, set the rowselection attribute to false in the datagrid tag on the JSP page, for example:

```
<dmf:datagrid ..rowselection="false">
```

Note: To support row selection the <dmf:datagrid> tag must be placed outside any HTML table tags.

[Table 68, page 189](#) describes the interaction between the global row selection flag in app.xml and the datagrid attribute.

Table 68. Row selection settings

app.xml <rowselection>	dmf:datagrid rowselection	Result
false	true or false	Checkboxes rendered, no mouse/keyboard row selection or context menus on any datagrid
true	true	No checkboxes rendered, row selection and context menus enabled for the datagrid
true	attribute not specified (for example, migrated customizations)	No checkboxes rendered, row selection and context menus enabled for the datagrid
true	false	Checkboxes rendered, no mouse/keyboard row selection or context menus on current datagrid

Configuring multiple rows for the same data object — Some data rows should be displayed as multiple HTML table rows. For example, the search results display a summary that cannot be selected. When the user selects one of the data rows in the set of table rows, the entire set should be selected. For these rows, add a datagridRowBreak tag at the position for which a new row should start. This will generate a </tr></tr> sequence that closes the current HTML and starts a new one. This is accomplished as follows in the search results JSP page when a summary is displayed a search result item:

```
<dmf:datagridRowTd width="99%"></dmf:datagridRowTd>
<!-- Display summary row if a summary is available -->
<dmf:panel datafield="summary">
<dmf:datagridRowBreak/>
<dmf:celllist>
<dmf:celltemplate field="summary">
```

```

<td colspan="23" class="additionalRow">
<dmf:termshighlightingformatter datafield='<%=SearchResultSet.ATTR_THH%>'
  cssclass='termshighlight' separator='<%=SearchResultSet.THH_SEPARATOR%>'
  visible='<%=String.valueOf(form.isHighlightActive())%>'>
  <dmf:stringlengthformatter maxlen='130'>
    <dmf:label nlsid="MSG_ATTR_SUMMARY" style="font-weight:bold"/>:
    <dmf:label datafield='summary' />
  </dmf:stringlengthformatter>
</dmf:termshighlightingformatter>
</td>
</dmf:celltemplate></dmf:celllist></dmf:panel></dmf:datagridRow>

```

Customizing row selection

Handling links within a datagrid row — To disable links within a datagrid row, such as `actionbutton`, `actionlink`, `actionbutton`, `button`, or link tags, surround the cell in the row with a `datagridRowModifier` tag. Then the link will be displayed only when row selection is turned off. To migrate links and action controls within a datagrid row with row selection turned on, move them to a context menu or a `datagridRowEvent` tag. (Refer to [Configuring context \(right-click\) menus, page 134](#) for more information on context menus.)

Note: If your datagrid row has checkboxes and your component uses the checkboxes to maintain state, you can render the checkboxes but not display them when row selection is on. To do this, set the `skipbody` attribute on the `datagridRowModifier` tag to `false`.

Example 4-1. Hiding a link when row selection is turned on

The following example hides the properties icon (an `actionimage` tag) when row selection is on:

```

<dmf:datagridRowTd cssclass='doclistfilenamedatagrid'>
<dmf:datagridRowModifier>
  <dmfx:actionimage name='propact' nlsid='MSG_PROPERTIES'
    action='properties' src='icons/info.gif' showifdisabled='false'>
    <dmf:argument name='objectId' datafield='r_object_id' />
    <dmf:argument name='type' datafield='r_object_type' />
  </dmfx:actionimage>
</dmf:datagridRowModifier>
<dmf:datagridRowTd cssclass='doclistfilenamedatagrid'>

```

Example 4-2. Handling a double-click event

You can define a single-click or double-click event for a row using the `datagridRowEvent` tag.

Note: A `datagridrow` tag can contain only one `datagridrowevent` tag.

The `datagridRowEvent` tag can be used to wrap an existing link tag so the link selection is handled upon double-click by a client-side or component class event handler. In the following example, the **Object name** link is migrated to a double click event:

```

<dmf:datagridRowEvent eventName='dblclick' eventhandler='
onClickObject' runatclient='true'>
  <dmf:link onclick='onClickObject' name='
objectLink' runatclient='true' datafield='object_name'>

```

```

    <dmf:argument name='id' datafield='r_object_id' />
    <dmf:argument name='type' datafield='r_object_type' />
    <dmf:argument name='isFolder' datafield='isfolder' />
  </dmf:link>
</dmf:datagridRowEvent>

```

Example 4-3. Handling a single-click event (selection)

The following example handles a single-click event (selecting the row). The event handler for a select event must be client-side:

```

<dmf:datagridRowEvent eventname='select' eventhandler='
  onSelectObject' runatclient='true'>
  <dmf:argument name='id' datafield='r_object_id' />
  <dmf:argument name='name' datafield='object_name' />
</dmf:datagridRowEvent>

```

The event object that is passed to the JavaScript handler has the following properties:

- **type**
Event type, such as select or init
- **datagrid**
Datagrid object instance
- **startIndex**
Index of the selected or deselected item, starting with 1.
- **count**
Count of selected items.

An event handler gets the item arguments, either for single or multiple selection, similar to the following:

```

function onSelectObject(event)
{
  //for single item selection
  if (event.count == 1)
  {
    var args = event.datagrid.data.getItemActionArgs(
      event.startIndex, event.type);
    //handle args
  }
  //handle multiple select
  else
  {
    for (var i=0; i<event.count; i++)
    {
      var args = event.datagrid.data.getItemActionArgs(
        event.startIndex + i, event.type);
      //handle args
    }
  }
}

```

A client-side JavaScript object models the datagrid in order to maintain datagrid client-side state, register client-side event handlers, and interact with browser DOM to respond to UI events.

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 datagrid flow and folder refresh

The folders in a cabinet are cached for display. You can configure a cache timeout for folders in the file `FolderUtil.properties`, located in `WEB-INF/classes/com/documentum/web/formext/docbase`. The key `cacheLifetime` specifies the cache lifetime between refreshes in seconds. The default value is 600 (ten minutes). This can be changed if a faster refresh is needed.

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 number of columns of data in the row.

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 paging 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>
...

```

Configuring drop-down lists

Drop-down 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 drop-down 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:

- Fixed values that are provided at design time in the JSP page, for example:

```
<dmf:option value=5 nlsid="Five"/>
```

- Runtime 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>
```

To alter the drop-down list programmatically from your component class, set the mutable attribute to true (default = false).

- **datadropdropdownlist**

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:datadropdropdownlist name="<%=FormatAttributes.FORMAT_RENDITION_LIST%">"
  tooltipnlsid="LABEL_RENDITION">
  <dmf:dataoptionlist>
    <dmf:option datafield="name" labeldatafield="name"/>
  </dmf:dataoptionlist>
</dmf:datadropdropdownlist>
```

- Values that are provided by options, one or more of which is provided by a datafield. For example:

```
<dmf:datadropdropdownlist 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:datadropdropdownlist>
```

You can use a DQL query to set the results for a datadropdropdownlist control. The following example from a component class gets a list of formats from the repository and displays them for user selection in a drop-down 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);
  ...
}
```

Configuring attribute display

Individual attributes, both single-value and repeating, are displayed by 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.

List of attributes are rendered by the `docbaseattributelist` control. For information on configuring this control, refer to [Generating lists of attributes for display, page 196](#).

If a single attribute is editable, a text box 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 repeating attribute is editable, an **Edit** link that loads the repeating attribute editor is rendered.

If an attribute has value assistance, it is displayed by either the `docbasesingleattribute` or `docbaserepeatingattribute` component. These components can be extended and configured to present a different editing page for different object types.

Generating lists of attributes for display

You can generate 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 or format certain attributes or attribute types as generated by the `DocbaseAttributeListTag` class. Refer to [Modifying the display and handling of attributes, page 162](#) for details.

The following topics describe how to use the control and configure a list of attributes for various contexts:

- [Configuring the attributelist control, page 196](#)
- [Creating a context-based attribute list, page 197](#)
- [Creating an attributelist that is independent of the data dictionary, page 198](#)

Configuring 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 [Creating an attributelist that is independent of the data dictionary, page 198](#) for a description of the attribute configuration file. Refer to [Creating a context-based attribute list, page 197](#) 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.

Creating a context-based attribute list

Data dictionary lists of attributes can be created with Documentum Application Builder (DAB). For the procedure, refer to the DAB documentation. When you define a scope in DAB, you must have a matching qualifier in your application. For example, if you define a scope "type" in DAB, you must have the type qualifier enabled in your application app.xml file.

The lists of attributes can be limited based on user context such as group or folder location by creating a preset in Webtop.

Categories as defined in DAB are displayed in the WDK 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.

To use data dictionary attribute lists, set the value of the `<data_dictionary_population>` element to true (the default) in your `docbaseattributelist` configuration file. 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: If your object type has constraints, be sure to display the constrained attributes. If you display a constrained attribute with a control outside the `docbaseattributelist` control, you must list this attribute in the `<ignore_attributes>` element of the `docbaseattributelist` definition and then validate the control on the JSP page using the `docbaseobjectvalidator` control. In the following example from the import JSP page, the format (`a_content_type`) is validated with this control:

```
<dmf:docbaseobjectattributevalidator name="attrvalidator" object="docbaseObj"
  controltovalidate="formatlist" attribute="a_content_type"/>
```

The data dictionary scopes that you define in Documentum Application Builder might look like the following:

Table 69. 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.

Creating an attributelist that is independent of the data dictionary

The attribute configuration file 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 by overriding 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

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.

You can extend a definition using the same `extends` attribute that is used for other WDK configuration files. (Refer to [Extending XML definitions](#), page 40.)

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.

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.

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 name=attribute_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. Set to false to use the values in the configuration file. The client display does not write back to the data dictionary.
- 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 can 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. 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.

Configuring a "Starts with" filter

Datagrid items can be filtered using the **Starts with** filter, which is generated by a `datacolumnbeginswith` control within a `datagrid` tag or within a `datadropdownlist` on the JSP page. Filtering is case-insensitive by default. In the following example from `myobjects_list_body.jsp`, the filter displays a text box that filters on object name starting with the text entered by the user.

```
<dmf:datacolumnbeginswith name='namecolumnbeginswith' column='
  object_name' autocompleteid='ac_namecolumnbeginswith' size="
  24" nlsid="MSG_BEGINSWITH_FILTER"/>
```

The **Starts with** filter can be applied to either a `datagrid` or a `datadropdownlist`. The filter will be bound to the containing control on the JSP page that implements `IDataboundControl`. In the following example from a JSP page, the `datacolumnbeginswith` control is bound to the `datadropdownlist`:

```
<dmf:datagrid ...>
  <dmf:datadropdownlist ..>
    <dmf:datacolumnbeginswith ../>
  </dmf:datadropdownlist ..>
</dmf:datagrid>
```

The data source that will be filtered is specified in the `column` attribute of the `datacolumnbeginswith` control. The column must match a column that is specified within a `<column>` element of the component definition. For example, to enable filtering on the object name, the value of the `column` attribute should be `"object_name"`.

Note: The data provider instantiated by the component class issues a query for column data. If the query is `"select object_name AS NAME..."` then the value of the `column` attribute on the `datacolumnbeginswith` control should be `"NAME"`. An error will be thrown if the value of `"column"` cannot be mapped to any attributes from the data provider.

The `column` attribute is overridden by the `datafield` attribute, which must match a `datafield` specified in the `datagrid`. The `datafield` can match a dynamic column named `CURRENT`. The `datafield` attribute can be used inside a `celltemplate` tag for dynamic binding to a rendering column. For instance, to add a filter control in front of a sort control, add the following in the JSP:

```
<dmf:celltemplate>
<th scope="col" align="left" class="doclistfilenamedatagrid">
  <nobr>
    <dmf:datacolumnbeginswith name="filter6" datafield="CURRENT"/>
    <dmf:datasortlink name="sort6" datafield="CURRENT"/>
  </nobr>
</th></dmf:celltemplate>
```

This control can be optionally enclosed within a formatter to allow filtering based on display values. For example, a WDK list view uses a formatter to translate the internal content type `msw8` to the display value `Microsoft Word document` as shown below:

```
<dmf:celltemplate field="a_content_type">
<td nowrap class='doclistfilenamedatagrid'>
  <dmf:stringlengthformatter maxlen="14">
    <dmfx:docformatvalueformatter>
      <dmf:label datafield="CURRENT"/>
    </dmfx:docformatvalueformatter>
  </dmf:stringlengthformatter>
</td></dmf:celltemplate>
```

To enable filtering on the displayed content types, add the filter control as follows:

```
<dmfx:docformatvalueformatter>
  <dmf:datacolumnbeginswith column="a_content_type">
  </dmf:datacolumnbeginswith>
</dmfx:docformatvalueformatter>
```

Filtering on repeating attributes is not supported.

Providing data to databound controls

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 `datadropdownlist` control 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 attributes 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);
    }
}
```

Accessing datagrid controls

You can iterate through the rows of a datagrid and get values of the contained controls using `Control.getContainedControls()`.

Example 4-4. Getting a value from a datagrid row

The following example is taken from code that fetches a datafield value from a datagrid 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
                }
            }
        }
    }
}
```

Getting data in a component class

If your component JSP pages contain databound controls, you must instantiate a `DataProvider` class to provide data to the controls. 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 also get data for your behavior class using `DfQuery`.

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`.

The `setQuery()` method of `DataProvider` 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 from a query

Your component or action class may need to run a query that does not provide data directly to a databound control. (Databound controls get data from the data provider in the component class.) 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 `TableResultSet`. (`IDfCollection` objects can also provide information directly to text, textarea, and dropdownlist controls.)

Note: If you use an Oracle JDBC result set to construct a table result set, the column names generated are uppercase.

Example 4-5. Running a query

The following example from the `NewRoom` class runs a query using `DfQuery`. The results are returned in an `IDfCollection` and put into a `TableResultSet` for display in a datagrid.

```
TableResultSet queryRoomTypes (String strLanguage)
{
    TableResultSet resultSet = new TableResultSet(new String[] {
        "type_name", "label_text"});

    //Ask data dictionary for types whose supertype is dmc_room
    StringBuffer buf = new StringBuffer(256);
    buf.append("select b.type_name, b.label_text from
        dmi_type_info a, dmi_dd_type_info b where")
        .append(" (any a.r_supertype = '').append( Constants.DMC_ROOM )
        .append(") and (a.r_type_name = b.type_name)")
        .append(" and (b.nls_key = '').append( strLanguage )
        .append(") order by 2,1");

    try
    {
        IDfQuery query = DfcUtils.getClientX().getQuery();
        query.setDQL(buf.toString());
        IDfCollection iCollection = query.execute(
            getDfSession(), IDfQuery.READ_QUERY);

        try
        {
            while ( iCollection.next() )
            {
                String strTypeName = iCollection.getString("type_name");
                String strLabel = iCollection.getString("label_text");
                String strDescription = strLabel + " (
                    " + strTypeName + ")";
                resultSet.add(new String[]{strTypeName, strDescription});
            }
        }
        finally
        {
            iCollection.close();
        }
    }
}
```

```
catch (DfException e)
{
    throw new WrapperRuntimeException("
        Unable to query folder types from docbase!", e);
}
return resultSet;
}
```

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 IDfCollection object, XML file or JDBC result set.

Example 4-6. 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 drop-down 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 to get data in the JSP page:

- By overriding a query for a databound control, if the control supports this attribute
- By getting data from the data provider

Example 4-7. 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 4-8. Getting data from a data provider

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 `inbox_list_body.jsp` in `webcomponent/navigation/inbox`, the data provider gets the due date and displays overdue items differently. Note that the JSP page must import the `DataProvider` class:

```
<%@ page import="com.documentum.web.form.control.databound.Datagrid"%>
<%@ page import="com.documentum.web.form.control.databound.DataProvider"%>...
<% String due_date = provider.getDataField("due_date");
   if(due_date == null || due_date.length() == 0 ||
      Long.parseLong(due_date) > System.currentTimeMillis())
   {%>
     <dmf:label datafield='due_date' />
<% }
   else
   {%>
     <dmf:label style='COLOR:red' datafield='due_date' />
<% }%>
```

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 4-9. 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` located in `WEB-INF/classes/com/documentum/we/.form/control/databound`. 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 4-10. 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.

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. Then you set the result set for the databound control (the control that will display the data) using `DataProvider.setResultSet()`.

The base resultset 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.

Example 4-11. Setting data to a databound control

You can call `setScrollableResultSet()` on `DataProvider` to set data in a databound control.

In the following example from `ChangeHomeDocase`, a `ListResultSet` is set by the `DataProvider` to a data drop-down list control:

```
DataDropDownList docbaseList = (DataDropDownList) getControl(
    DOCBASE_LIST, DataDropDownList.class);
docbaseList.getDataProvider().setDfSession(getDfSession());

ListResultSet docbaseListResultSet = new ListResultSet(
    getDocbaseNames(), "docbasename");
...
docbaseList.getDataProvider().setScrollableResultSet(docbaseListResultSet);
```

`ConfigResultSet` gets data from an XML configuration file. Refer to [Passing Data from a Vector to a TableResultSet](#), page 209 for an example of putting configuration file content into a `Vector` and passing the data to a table or list.

Example 4-12. Passing data from a configuration file to a data drop-down 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);
```

Example 4-13. 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 4-14. 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 4-15. Passing Data from a Vector to a ListResultSet

In the following example from UserImport, getDocbaseNames() provides a Vector of repository names for drop-down 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);
```

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 data columns 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 data handler can implement the method `processDataFilterSpec()` to handle data filtering by a `datacolumnbeginswith` control. To support custom filtering requirements, such as a filter based on "Contains" rather than "Begins with" or a set of complex filtering criteria, `DataFilterSpec` can be extended.

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 navigation 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 4-16. 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`.

Supporting lightweight sysobject display in a component

You can add support to a listing component to display the parent of lightweight sysobjects (LWSOs). For information on how to configure parent display in listing and locator components, refer to [Configuring lightweight sysobject parent display, page 97](#).

In the `updateControlsFromPath` method, get the datagrid control and then add the DQL hint that hides LWSO parents as follows:

```
//import LWSO utility class
import com.documentum.web.formext.docbase.LightWeightSysObjectUtil;

protected void updateControlsFromPath(String strFolderPath)
{
```

```

...
//existing code to set query on datagrid
Datagrid datagrid = ((Datagrid)getControl(CONTROL_GRID, Datagrid.class));

//Add the DQL hint to hide LWSO parents
strQuery = LightweightSysObjectUtil.addHideSharedParentHintToQuery(findByIDQuery);
//end add DQL hint

datagrid.getDataProvider().setQuery(strQuery, findByIdQuery);

```

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 value assistance independent of the data dictionary only 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 value assistance control, refer to *Web Development Kit and Webtop Reference*.

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. For example:

```

<dmfx:docbaseattribute object="obj" attribute="keywords"
  docbaseattributevalueassistance="my_set" 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_set" 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 assistance independent of the data dictionary. For example:

```
<dmfx:docbaseattribute object="obj" attribute="keywords"
  docbaseattributevalueassistance="my_set" 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_set" 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);
```

Configuring and Customizing Actions

The following topics describe common configuration and customization tasks for actions as well as general information on how actions work:

- [Actions overview, page 215](#)
- [Configuring actions, page 216](#)
- [Configuring and customizing shortcuts to actions \(hotkeys\), page 222](#)
- [Customizing actions, page 227](#)

Actions overview

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 does not require a UI, such as the logout action.

A preset can be used to limit the actions that are available to a user in a specific group, location, or on a specific object type.

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 [Configuring an action definition, page 219](#)). 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 40](#).

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 216 for details.

Actions can be role-based. Refer to [Chapter 10, Configuring and Customizing Roles and Client Capability](#) for further information on role-based actions.

Turning on action tracing — Set the following flag key 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.

Configuring actions

Actions can be called by URL or from a component class. Actions can also be called when a user clicks an action-enabled control in the UI. 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 is launched by URL, the WDK framework searches for the action definition by action ID, evaluates actions preconditions, and executes the action. For information on the action attributes that configure action controls, refer to *Web Development Kit and Webtop Reference*. For information on the parameters and configurable elements for each action, refer to *Web Development Kit and Webtop Reference*.

Actions can be limited by user context such as role or location in the repository by creating a preset. For information on creating a preset for actions, refer to *Webtop User Guide*.

The following topics describe general action configuration and implementation.

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 reference by `<%@ include file='doclist_thumbnail_body.jsp' %>` is in the same directory as the including file. The included file reference by `<%@ include file='/wdk/container/modalDatagridContainerStart.jsp' %>` is in the wdk 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 121](#).

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 5-1. 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 5-2. 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 5-3. 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
...

```

Configuring an action definition

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 or modified in the same way that application and component definitions are extended and modified.

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" requires 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 you use `LaunchComponent`, you can add the optional child elements `<navigation>`, `<component>`, and `<container>`. If a `<component>` element is not present, the `LaunchComponent` class will use the action name for the ID of the component to be launched.

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>
```

Removing an action from a component

Certain actions may not be appropriate for a particular component. To remove an action for a component, use the `ComponentPrecondition` class in an action definition. The `<reverse-component>` element specifies the component or components for which the action will not be available. In the following example from `dm_sysobject_clipboard_actions.xml`, the `move` action is disabled for the `search` component:

```
<action id="move">
...
<preconditions>
  <precondition class="com.documentum.webcomponent.environment.actions.MoveAction">
  </precondition>
  <precondition class="com.documentum.web.formext.action.ComponentPrecondition">
    <reverse-precondition>true</reverse-precondition>
    <component>search</component>
  </precondition>
</preconditions>
<execution class="com.documentum.webcomponent.environment.actions.MoveAction">
```

```
</execution></action>
```

Hiding or displaying an action

You can hide actions in specific locations of the repository or for object types or user groups by creating a preset.

By default, all invalid actions are hidden in menus, based on the setting of the global flag `<hideinvalidactions>` in `app.xml`. To display invalid actions, if they are not hidden by a preset, you can add a modification file to `custom/config` with the following contents:

```
<application modifies="wdk/app.xml">
  <replace path="display.hideinvalidactions">
    <hideinvalidactions>false</hideinvalidactions>
  </replace>
</application>
```

Note: Because the `<application>` element that you are modifying does not have an ID, you must leave it out of the `modifies` attribute value as shown in the example.

If an action definition is scoped to a specific object type, the definition applies to subtypes of the type. To turn off the action for certain subtypes, 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 those actions cannot be performed on reference objects. This list of undefined actions is combined with a dynamic filter on the action definition so foreign objects display an invalid action message when the user selects the action. Refer to [Using dynamic filters in action definitions](#), page 233 for details on the filter.

Configuring and customizing shortcuts to actions (hotkeys)

Shortcuts, also called hotkeys, are key combinations that can be configured to call WDK actions. Shortcuts are enabled for the entire application in `app.xml`. For information on enabling or disabling shortcuts globally in your custom `app.xml` file, refer to [Enabling shortcuts \(hotkeys\)](#), page 80.

Shortcuts are supported for the following controls: `button`, `link`, `menuitem`, `actionbutton`, `actionlink`, `actionmenuitem`, and `actionimage`. Custom controls that extend these control tags inherit `shortcut`

support if they add the appropriate attributes to the tag library descriptor. For information on these attributes, refer to [Adding shortcut support to a control, page 226](#).

Controls that support shortcuts have a `hotkeyid` attribute, for example:

```
<dmfx:actionmenuitem... action='checkin' hotkeyid='HOTKEY_CHECKIN' />
```

The `hotkeyid` value is resolved by a lookup in the shortcuts (hotkeys) definition in `hotkeys.xml`. This file defines an NLS key for each hotkey ID. The key is then resolved to a key combination in the hotkeys properties file `HotKeysNlsProp.properties`. The properties files can be localized for locale-specific key combinations.



Caution: When focus is on a user-entry control such as text, shortcuts are not enabled. If you set initial focus in the UI to a user-entry control, shortcuts will not be enabled until the user moves off the control.

To create your own shortcuts mapping, perform the following tasks:

1. Specify your hotkeys mapping file in `app.xml`. Refer to [Specifying a shortcuts mapping file, page 223](#).
2. Create a definition by extending or modifying `hotkeys.xml`. This file will have a hotkey ID for each hotkey action. Refer to [Creating a shortcut definition, page 223](#).
3. Create a map that maps each hotkey ID to a key combination. Refer to [Creating a shortcuts map, page 224](#)

Specifying a shortcuts mapping file

The `<hotkeys>` element in your custom `app.xml` file specifies a mapping file that maps key combinations to actions. You must create a Java properties file similar to the WDK mapping file, `HotKeysNlsProp.properties`. The WDK properties file is located in `webcomponent/strings/com/documentum/webcomponent/keyboardshortcut`. If your file is located in `WEB-INF/com/mycompany`, for example, you would specify the location as follows in `app.xml`:

```
<hotkeys>
  <enabled>true</enabled>
  <nlsbundle>com.mycompany.HotKeysNlsProp</nlsbundle>
</hotkeys>
```

Creating a shortcut definition

The `hotkeyid` value for a control is resolved by a lookup in the hotkeys configuration file `hotkeys.xml`, located in `webcomponent/config`.

Tip: If you are changing existing shortcut combinations, you do not need a hotkeys definition. If you are adding shortcuts for your custom actions, a hotkeys definition is required. You can use the modification mechanism to do this.

To create your own hotkeys definition, create a modification XML file on custom/config, for example, hotkeys_modification.xml, that modifies the WDK hotkeys definitions and adds custom keys. For example:

```
<hotkeys modifies="hotkeys:webcomponent/config/hotkeys.xml">
  <insert>
    <hotkey id=...>
  </insert></hotkeys>
```

Table 70, page 224 describes the hotkeys configuration elements.

Table 70. Hotkeys configuration elements

Element	Description
<hotkeys id=...>	The id attribute on this element facilitates more than one hotkeys definition for the application
<hotkey id=...>	The id attribute on this element is referenced by a control on the JSP page and contains a key for lookup of the hotkey combination. The ID must be unique to the definition. Contains <keynlsid>.
<keynlsid>	Specifies an NLS key that is resolved in the hotkeys NLS properties file.
<labelnlsid>	(Optional) Specifies an NLS key for a label that is different from the hotkey command string

Creating a shortcuts map

A Java NLS properties file specifies the key combinations for each hotkey ID. Specify the fully qualified pathname for this NLS properties file in your custom app.xml file. If you add actions to the hotkeys definition, then create NLS IDs for your shortcut IDs in a custom hotkeys configuration file.

To change or add hotkey combinations, include the WDK properties file in your own properties file as follows:

```
NLS_INCLUDES=com.documentum.webcomponent.keyboardshortcut.HotKeysNlsProp
```

In your key combinations, a single keyboard key can be combined with another key such as Ctrl (Windows), Cmd (Macintosh), Shift, or Alt. Table 71, page 225 describes the single keys that can be used in a shortcut combination. Shortcut definitions are case-insensitive.


```

    </hotkey>
</insert>
<replace>

```

6. Create a `HotKeysNlsProp.properties` file in `WEB-INF/classes/com/mycompany`. The following example adds an entry for the new hotkey ID and maps it to the key combination of the alt key and the x key. It changes the shortcut for the copy action:

```

NLS_INCLUDES=com.documentum.webcomponent.keyboardshortcut.HotKeysNlsProp
_#HOTKEY_MYACTION=Alt+X
_#HOTKEY_ADD_TO_CLIPBOARD=Shift+X

```

7. Add the `hotkeyid` attribute value `HOTKEY_MYACTION` to the component JSP page, in the control that calls your custom action, for example:

```

<dmfx:actionlink name="mylink" action="myaction" hotkeyid="HOTKEY_MYACTION"...>

```

8. Restart the application server for changes to NLS properties files. If your changes are to XML files only, refresh memory by navigating to `wdk/refresh.jsp`.
9. Test your shortcut combination in the appropriate component.

Shortcuts are bound to the top level window of the application, so you must ensure that all shortcuts are uniquely defined in the `<hotkeys>` configuration elements across your application. This will be easier to manage in a single shortcuts mapping file. If two controls are assigned the same shortcut, the second assignment will be used. Shortcuts are not invoked when the keyboard focus is on an input field such as text, textarea, or password. For these controls, use the escape key to access shortcuts.

Adding shortcut support to a control

The base `Control` class supports shortcuts with the public methods `setHotKey`, `setHotKeyLabel`, `getHotKey`, and `getHotKeyLabel`.

The following WDK controls support shortcuts: `ActionMenuItem`, `ActionButton`, `ActionLink`, `ActionImage`, `Button`, `Link`, and `MenuItem`. If your custom control extends a WDK control with shortcut support, add the `hotkeyid` attribute to the tag library descriptor that contains your control and then set the `hotkeyid` value on the control in the JSP page.

To set a shortcut combination programmatically on a control that supports shortcuts, use the API `setHotkey(String hotKey)` where the parameter is a key combination, for example:

```

setHotkey("alt+x");

```

If your control does not extend a control with shortcut support, add support to the tag class by calling either the `renderHotKey()` or `renderHotKeyHandler()` from `renderEnd()`.

To add shortcut support to a custom control

1. Add a `hotkeyid` attribute for the custom control to your control tag library descriptor as follows:

```

<attribute>

```

```

    <name>hotkeyid</name>
    <required>>false</required>
    <rtexprvalue>>true</rtexprvalue>
  </attribute>

```

2. At the end of `renderEnd()` in the control tag class, add a call to `renderHotKey()` or `renderHotKeyHandler()`.

Use `renderHotKey(StringBuffer buf, String eventName)` if the shortcut can be associated directly with a UI event on the control, such as the `onclick` event. Provide the event name, and the Javascript code for the shortcut will be rendered by the framework. The following example from `ButtonTag` checks whether the control is enabled and, if so, adds the `renderHotKey` JavaScript function key to the output. In the `Button` class, the `EVENT_ONCLICK` is the `onclick` event:

```

if (button.isEnabled())
{
    StringBuffer buf = new StringBuffer(256);
    renderHotKey(buf, Button.EVENT_ONCLICK);
    out.print(buf.toString());
}

```

Use `renderHotKeyHandler(StringBuffer buf, String hotkeyHandler)` for controls such as action controls in which the shortcut is associated with an action, not a UI event on the control. Specify a hotkey JavaScript handler that will be invoked for the shortcut and provide handling in the referenced JavaScript handler. In the following example from `ActionButtonTag`, a handler method `getClientEventFunctionCall` is specified (provide your own method):

```

if ((isButtonGraphic(button) && button.isEnabled(
)) || !isButtonGraphic(button))
{
    renderHotKeyHandler(buf, getClientEventFunctionCall());
}

```

Note: The handler `getClientEventFunctionCall` is implemented in `ActionControlTag` to pass on the dynamic attribute value of an action control. If your control is not an action control, you can call a custom handler in your tag class to render your JavaScript reference.

3. Follow the steps in [To add or change a shortcut combination, page 225](#) to set up and test the shortcut for your control.

The keyboard events that are handled are `keyup`, `keydown`, and `keypress`. WDK JavaScript handlers for each of these keyboard events retrieve the shortcut commands and invoke the corresponding actions.

Customizing actions

Common action customization tasks are described in the following topics.

Refer to [Configuring actions, page 216](#) for information on configuring actions. For information on the parameters and configurable elements for individual actions, refer to *Web Development Kit and*

Webtop Reference. For information on available precondition classes, refer to [Commonly used action preconditions, page 238](#).

Passing arguments to an action

Argument tags pass arguments to actions on selected objects as well as to the components that are launched by those actions. Action controls that do not support multiple selection, such as `actionlink`, can contain argument tags.

For actions that support multiple selection, the `actionmultiselect` tag passes arguments to the selected action. Refer to [Passing arguments to menu action items, page 229](#) for more information and an example of getting the argument in the component class that is instantiated by the action.

You can pass arguments to an action precondition class.



Caution: Do not add arguments to the action in a precondition class `queryExecute()` method. To add arguments in the action class programmatically, add them in the `execute()` method.

Example 5-4. 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>
  ...
</dmf:datagrid>
</dmfx:actionmultiselect>

<dmfx:actionbutton action='checkin' dynamic='true'
  nlsid="MSG_CHECKIN"/>
```

Note: You pass arguments through the `actionmultiselectcheckbox` control even when row selection is enabled and checkboxes are not used.

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");
}
```

Passing arguments to menu action items

Menu items do not pass arguments. Arguments are passed to the menu action through an `actionmultiselect` control. This tag is also used to pass arguments to an action that does not have a selected object, such as `import`. An action may use only some of the arguments in the `actionmultiselect` tag, because it contains arguments for all possible multiselect actions on the page. Arguments can also be passed within an `actionmultiselectcheckbox` tag, but those arguments will not be passed to `genericnoselect` actions, that is, actions that ignore selected objects.

Note: You can pass arguments through the `actionmultiselectcheckbox` control even when row selection is enabled and checkboxes are not used.

In the following example, a menu item in `menubar_body.jsp` specifies the `import` action:

```
<dmfx:actionmenuitem dynamic='genericnoselect' name='file_import' .../>
```

The `import` action definition in `dm_folder_actions.xml` defines several parameters for `import`, of which only the target folder object ID is required. The required argument is passed in `doclist_body.jsp` within an `<actionmultiselect>` tag as follows. (The other arguments are used by other actions.)

```
<dmfx:actionmultiselect name='multi'>
  <dmfx:argument name='objectId' contextvalue='objectId' />
  <dmfx:argument name='type' contextvalue='type' />
  ...
</dmfx:actionmultiselect>
```

The arguments are passed to the action class that launches the action. The action class may use the arguments for precondition or execution evaluation or pass the arguments on to the component. For example, the `import` component gets the arguments of the object ID, which is the target folder for `import`, as follows:

```
setFolderId(arg.get("objectId"));
```

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. 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>
```

Note: If row selection is turned on and checkboxes are not visible, you must still pass arguments in an `actionmultiselectcheckbox` or `actionmultiselect` control.

If your action is of the `genericnoselect` type, pass the argument in the `actionmultiselect` control, not the `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.

Getting arguments in the component — The arguments are passed to the action class that launches the multiple action. The action class passes the arguments to the component in an `ArgumentList` instance. 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");
  ...
}
```

To get multiple argument values that are passed by actions that support multiple selection, call the `ArgumentList` method `getValues()`. The following example from the `AclValidate` class `onInit()` puts the values into a `String` array, stores them in a `Hashtable`, and accesses them individually:

```
public void onInit(ArgumentList args)
{
  // call the super component to perform the setup work
  super.onInit(args);
  String strComponentArgs[] = args.getValues("validation");
  if(strComponentArgs.length > 0)
  {
    m_userMap = new Hashtable();
  }
  for(int index = 0; index < strComponentArgs.length; index++)
  {
    String encodedArgs = strComponentArgs[index];
    ArgumentList userValidationList = ArgumentList.decode(encodedArgs);
    String userName = userValidationList.get("username");
    String permitType = userValidationList.get("permitfailed");
    String groupName = userValidationList.get("groupname");...}
}
```

Using the 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.

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 webcomponent/config/actions/generic_actions.xml.

Tip: The selected object ID for a component, 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 [Adding an action listener](#), page 241. 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>
```

Creating a custom action definition

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 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>

```

Using dynamic filters in action definitions

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. For most purposes, the `LaunchComponentFilter` is sufficient to launch a component after your evaluator class evaluates the context.

The filter uses an evaluator 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. The evaluator class returns true when a criterion is matched, and then the filter class launches the specified component. For example, the versions action uses a `ReferenceEvaluator` class to evaluate the criterion "isreference". The `evaluate()` method of the evaluator class is passed the argument list of the containing component, gets the object ID argument, and tests the value of the `i_is_reference` attribute of the object. The method then returns true for reference objects, and the filter launches the specified component as shown in the action definition:

The contents of the filter element are similar to the following:

```

1 <dynamicfilter class='com.documentum.web.formext.action.LaunchComponentFilter'>
2 <option>
3 <criteria>
4   <criterion name='isreference' value='true'
   evaluatorclass='com.documentum.webcomponent.library.actions.ReferenceEvaluator' />
5 </criteria>
6 <selection>
7   <component>informinvalidactionforreference</component>
8 </selection>
9 </option>
10 </option>
11 <criteria></criteria>
12 <selection>

```

```

    <component>versions</component>
    7<container>navigationcontainer</container>
    8<navigation>jump</navigation>
  </selection>
</option>
</dynamicfilter>

```

- 1 Specifies the filter class and contains two or more options (<option> elements) for launching different components from the same action
- 2 Defines an option that specifies the component that will be launched when criteria are met. You must provide at least two options for the filter. Your evaluator class must return the value specified as <criteria ...value=some_value> in order for the filter class to launch the component named in <selection>.<component>.
- 3 Contains zero or more <criteria> elements. If this element is empty, the selection is the default selection.
- 4 Defines a criterion that must be matched. 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
- 8 Specifies the navigation to the specified container. Valid values: jump | returnjump | nested (default).

Example 5-5. Implementing a dynamic evaluator class

This example uses the LaunchComponentFilter to launch a component based on the evaluation of a criterion. The evaluator class tests for the current repository and launches one component for true, another component for false.

The example modifies the delete action definition for dm_folder to add a new criterion with a custom evaluator class. The definition that is being modified does not have a filter, so we add this element to the <execution> element. The default behavior from the original definition will occur if the filter criteria returns false. (We do not need to name the component within combocontainer, because the LaunchComponent class will use the name of the action to resolve the name of the component.) If the filter returns true, the about component will be launched.

```

<action modifies="
  delete:application="webcomponent" type="dm_folder">
  <insert path="execution"
    <dynamicfilter class="
      com.documentum.web.formext.action.LaunchComponentFilter">
    <option>
      <criteria>
        <criteria name='istest' value='true' evaluatorclass='
          com.mycompany.TestEvaluator' />
      </criteria>
    <selection>
      <component>about</component>
    </selection>
  </option>

```

```

    <option>
      <criteria></criteria>
      <selection>
        <container>combocontainer</container>
      </selection>
    </option>
  </dynamicfilter>
</insert>
</action>

```

The evaluator class implements `ILaunchComponentEvaluator`, gets the current repository, and returns true if the criterion is matched:

```

package com.mycompany;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.common.DfException;
import com.documentum.web.formext.action.ILaunchComponentEvaluator;
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;

public class TestEvaluator implements ILaunchComponentEvaluator
{
  public String evaluate(String strName, String strAction, IConfigElement config,
    ArgumentList arg, Context context, Component component)
  {
    System.out.println("In the TestEvaluator class");
    String strValue = "false";

    if (strName != null && strName.equalsIgnoreCase(CRITERION))
    {
      try
      {
        IDfSession session = component.getDfSession();
        if (session.getDocbaseName().equals("test_repository"))
          System.out.println("Repository is test_repository");
        {
          strValue = "true";
          System.out.println("returning true");
        }
      }
      catch (DfException e)
      {
      }
    }
    return strValue;
  }
  public static String CRITERION = "istest";
}

```

Creating an action precondition

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.

Whether to use an action precondition or an action preset — Use the following guidelines to decide whether a precondition or action preset is better:

- Simple context

For a precondition that applies to a particular user context such as object type, location, or group, you can use a preset instead of precondition. A precondition can contain more logic than a preset to determine whether an action should be available.

- Run-time vs. design-time constraint

Preconditions are set at design time by the developer. Presets are set at runtime by the business user.

An action must pass both the preset and the precondition checks in order to be considered valid in the current context. The preset is checked first by the ActionService. If the preset allows the action in the current context, then the precondition is run. If the preset does not allow the action, then the precondition is not even run. The action is marked as invalid in the current context.

Implementing a precondition — A 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 username 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.

Tip: 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. You can configure the application to test action preconditions only when they are executed instead of on page rendering. Set the onexecutiononly attribute of the precondition element to true as follows:

```
<precondition onexecutiononly="true" 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:

- *strAction* is the action name.
- *config* represents the associated <precondition> element and subelements. Use this if configuration information is passed to the precondition.
- *arg* is the list of arguments passed to the action
- *context* is the user's current context as defined by the component
- *component* is an optional argument for the caller component. Allows action precondition methods to access the container component instance if necessary.



Caution: A precondition class has the following limitations:

- Do not put into the `queryExecute()` method of the precondition class any code that sets state. Use the precondition to disable behavior in the UI, or put error handling code in the action `execute()` method.
- Avoid calling `IDfSession.getObject()` or performing 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.

Example 5-6. 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;
}
```

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.

Commonly used action preconditions

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>
```

The following precondition classes are used by multiple components and may be useful in your custom application.

Package `web.formext.action` —

- `ActionDisablerPrecondition`

Disables an action when this class is specified as the precondition class in an action definition. In the following example, the `locations` action is disabled for the object type `dm_cabinet`:

```
<scope type="dm_cabinet">
  ..
  <action id="locations">
    ...
    <preconditions>
      <precondition class="
        com.documentum.web.formext.action.ActionDisablerPrecondition"/>
    </preconditions>...
  </action>
</scope>
```

- `ComponentPrecondition`

Specifies one or more components for which the action does not apply. Component ID is specified as the value of the `<reverse>` child element. In the following example from the `move` action definition, the action is disabled for the `search` component:

```
<precondition class="com.documentum.web.formext.action.ComponentPrecondition">
  <reverse>true</reverse>
  <component>search</component>
</precondition>
```

- `GenericRolePrecondition`

Makes the action valid only for the specified role. The role must be one of the generic, client capability roles. In the following example, the `startworkflow` action is available only to users who are assigned a `contributor` role:

```
<precondition class="com.documentum.web.formext.action.GenericRolePrecondition">
  <role>contributor</role>
</precondition>
```

- RolePrecondition

Makes the action valid only for the specified role. The role can be any user-defined role. In the following example, the newcabinet action is available only to user who are assigned a coordinator role:

```
<precondition class="com.documentum.web.formext.action.RolePrecondition">
  <role>coordinator</role>
</precondition>
```

Package `webcomponent.library.actions` —

- ArgumentExistsPrecondition

Verifies that an argument has been passed to the action before action execution validates the required argument.

- ArgumentNotEmptyPrecondition

Verifies that an argument with a value has been passed to the action before action execution validates the required argument.

Implementing action 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 [Using the LaunchComponent execution classes, page 231](#)). 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 5-7. 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. The following example from `SuspendLifecycleAction` raises non-fatal errors in a catch block:

```
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 services `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 [Adding an action listener, page 241](#).

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 5-8. 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 5-9. 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");
```

Adding an action listener

Actions can have several kinds of listeners in the component class that calls the action:

- Component oncomplete handler

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.

- CallbackDoneListener

Use this listener if your component class launches the action that you are listening to. CallbackDoneListener returns the completion arguments to your component.

- 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.

- 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 5-10. Implementing an onComplete() event handler

Action controls launch actions. The component that hosts the action supports an 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.

You can use the Map to get completion arguments such as new object IDs, for example, after an import or a copy action.

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 `<dmf:argument>` or `<dmfx:argument>` tags within the action tag. For example:

```

<dmfx:actionimage ...>
  <dmf: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 5-11. Implementing `IActionCompleteListener`

To trap an action that is not launched by your component class or by an action control, you must implement `IActionCompleteListener`. From the `Map` object that is returned, you can get returned objects such as the object IDs after a copy operation.

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 5-12. Using `CallbackDoneListener`

To register your listener class, you must extend the action execution class and override the `execute()` and `getRequiredParams()` methods.

Alternatively, 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));
                        ...
                    }
                }
            }
        }
    }
}
```

Example 5-13. 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

To nest actions so 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.

To get more information from the called action than a simple Boolean return, register your precondition or execution class as an action listener, so you can get returned completion arguments (refer to [Adding an action listener, page 241](#)).

Example 5-14. 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
    }
    ...
}
```

Pre- and post-processing actions

Following is the procedure for creating an action execution class that performs pre- and post-processing.

To implement pre- or post-processing of an action

1. Extend an existing action execution class. For example:

```
public class CustomActionExecutionClass extends
    SomeExistingActionExecutionClass
```

2. (Pre-processing only) 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. (Post-processing only) Implement an action complete listener class. For example:

```

class ActionListener implements IActionCompleteListener
{
    public ActionListener(IActionCompleteListener listener)
    {
        m_listener = listener;
    }
    private IActionCompleteListener m_listener;
}

```

4. (Post-processing only) 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 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 for the example 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));
}
}
```


Configuring and Customizing Components

The following topics provide information on configuring and customizing components in WDK:

- [Configuring components, page 249](#)
- [Customizing components, page 280](#)
- [Configuring and customizing containers, page 304](#)
- [Using Business Objects in WDK, page 318](#)

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.

Common tasks that are applied to components are described in the following topics. For information on configuring specific WDK components, refer to *Web Development Kit and Webtop Reference*. For information on configuring containers, refer to [Configuring and customizing containers, page 304](#).

Modifying or extending a component definition

You can modify a component definition to add, remove, or replace parts of the WDK component definition. You can also extend a component to inherit the configuration of the parent component and override part of the definition. For simple modifications, use the `modifies` attribute instead of the `extends` attribute. Modifying a definition is preferable to extending it, because your modifications will benefit from changes to the parent definition on upgrade. For information on modifications to a component definition, refer to [Modifying configuration elements, page 33](#).

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 or modified 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. For more information on extending a definition, refer to [Extending XML definitions, page 40](#).

The strings for a component can be inherited from a component that you extend. Refer to [Inheriting strings, page 347](#) for details.

Versioning a component

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="5.3">`. If the version is not specified, it is the current version.

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 5.3 search component and has the same component ID, your custom component will be launched in place of the newer WDK 6 search component.

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

Table 72. Binding scenarios for versioned components

Custom layer	WDK layer	Binding
Extends 5.3 container of same ID	5.3 component loaded by 5.3 container (refer to note below)	Succeeds
Extends 5.3 component	5.3 container loaded by 5.3 action	Succeeds
Extends 5.3 container and component of same IDs	(refer to note below)	Succeeds

Custom layer	WDK layer	Binding
Extends 6.0 container and contains 5.3 component		Fails. Must extend 5.3 container to use 5.3 component or extend 6.0 container to use 6.0 component
Extends 5.3 container and contains 6.0 component		Fails. Must extend 5.3 component to use 5.3 container or extend 6.0 container to use 6.0 container.

Note: If your custom container extends a 5.3 container but has a different component ID, your container definition must have a <bindingcomponentversion> element with a value of 5.3.

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.

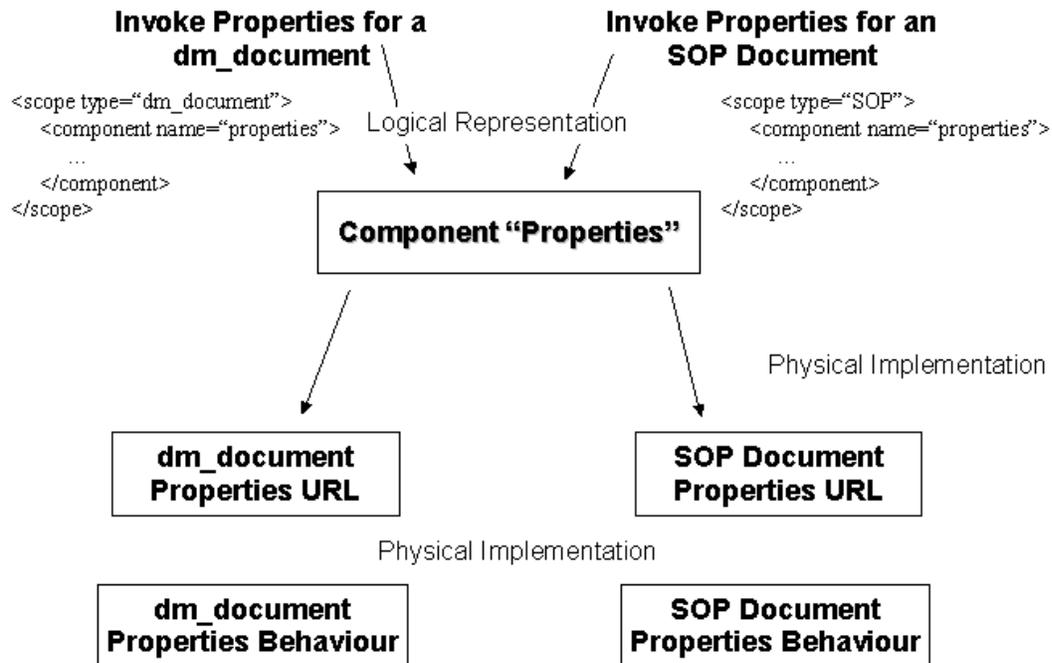
Scoping a component definition

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 14. 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 a component for specific contexts

You can hide component functionality in a specific context by creating a new definition for the context. You can also hide features within a component. For more information on filtering features, refer to [Hiding features within a component, page 253](#).

You can hide component functionality for contexts that match a qualifier value using the scope attribute on the `<component>` element. 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.

For example, you want to hide folder permissions. The original definition of the properties component has no scope attributes, so the the permissions component is displayed for all object types:

```
<contains>
  <component>attributes</component>
  <component>permissions</component>
  <component>history</component>
</contains>
```

Your modification definition is as follows:

```
<scope type="dm_folder">
  <component modifies="properties:webcomponent/config/library/properties/
    properties_component.xml">
    <replace path="contains">
      <contains>
        <component>attributes</component>
        <component>history</component>
      </contains>
    </replace>
  </component></scope>
```

You can hide component functionality for a defined scope using the `notdefined` attribute of the `<component>` element. 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>
```

The decision whether to hide a component using `<scope>`, `<filter>` or `<component 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. If you need to filter one feature within the component definition, use the `<filter>` element.

Hiding features within a component

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. Filter tags can be placed anywhere below a primary element.

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 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`.

The following example modifies the WDK properties component definition to show folder permissions to administrators only:

```
<scope type="dm_folder">
  <component modifies="properties:webcomponent/config/library/properties/
    properties_component.xml">
    <replace path="contains">
      <contains>
        <component>attributes</component>
        <component>history</component>
        <filter role="administrator">
          <component>permissions</component>
        </filter>
      </contains>
    </replace>
  </component></scope>
```

Calling a component 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` is the deployed application root context directory.

- *component_name* is the name of the component, including container components, as defined in the component definition XML file.
- *page_name* is the logical page name defined in the component definition XML file. If not present, the page defined by the <start> element is used.
- *params* are the 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 6-1. 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 6-2. Calling a component in a container by URL

Some components must be called within a container. Call the container component by URL and supply the contained component as a URL argument.

In the following example, the attributes component is called along with the object ID, so 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 a component from an action

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 6-3. 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 a component from an action class — You can jump to or nest to a component within an action class. In the action class `execute()` method, add your component parameters to the `ArgumentList` object and then the jump or nest. In the following example from `EditFormatPrefAction` class, the component name and arguments are added, then the component is nested to:

```
public boolean execute(String strAction, IConfigElement config,
    ArgumentList args, Context context,
    Component component, Map completionArgs)
{
    args.add("component", "format_pref_attr_selector");
    args.add(FormatPrefAttributesSelector.EDITING_PARAM_NAME, "true");
    component.setComponentNested("formatpreferencescontainer", args, context, (
        FormatPreferencesSummary) component);
    return true;
}
```

Calling a component from JavaScript

Client-side functions post navigation server events. Use client-side navigation functions to handle a client-side event that requires 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, 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();
}
```

Calling a component from another component

You can jump to or nest to another component from a component class by calling `setComponentJump` or `setComponentReturn`, respectively. If you are in a component that is in a container, call `setComponentReturnJump`. Add the relevant parameters to an `ArgumentList` object before

calling the component. The following example from the `ExtendedPermissions` class nests to the `acobjectlocatorcontainer` component when the user clicks **Change permission set**:

```
ArgumentList selectArgs = new ArgumentList();
selectArgs.add("component", "acobjectlocator");
selectArgs.add("flatlist", "true");
setComponentNested(
    "acobjectlocatorcontainer", selectArgs, getContext(),
    new IReturnListener()...);
```

Calling a component that launches a startup action

Any component can launch a startup action by adding the `startupAction` parameter to the URL or to the server-side method that calls the component. 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
```

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 and customizing containers, page 304](#). 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="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.

Configuring data columns in a component

Table 73, page 259 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.

Table 73. Data column configuration elements

Element	Description
<showfilters>	Displays the objectfilters drop-down control. Valid values: true false
<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>	Displays folders in specified format. 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.

Element	Description
<columns>. <loadinvisibleattribute>	Retrieves invisible attribute values in query. Uncomment this element and set to true to get invisible attribute values for use in your component. The invisible attributes can then be displayed by configuring a column in the <columns> element. Refer to Adding custom attributes to a datagrid, page 210 for details.
<columns><column>	Specifies columns to show or hide
<column><attribute>	Specifies the attribute to be displayed in the column.
<attribute> <label>	Sets a label for the column.
<column><attribute> <visible>	Displays the column. Valid values: true false

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='sortcoll' 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>
```

To display custom attributes, you must use dynamic columns. Refer to [Configuring dynamic data columns, page 262](#) for information.

To configure default columns for a component:

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>`. 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 6-4. 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>
</tr>
<!-- 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 runtime. 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.

Notes on column configuration:

- **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>
```

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 [Navigating within a component, page 280](#) 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 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.

All JSP pages that are not used by a component must contain a `<dmf:webform>` tag that specifies a form class, for example:

```
<dmf:webform formclass="com.documentum.web.form.Form"/>
```

Configuring 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 from the value of the nlsid attribute of a Documentum tag library element. Refer to [Configuring and customizing strings, page 346](#) 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 287](#).

Operating on a foreign object

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.

Processing a form before submission

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 172](#). If your component needs to do client-side processing before a JSP form is submitted, use the presubmission processing mechanism. For example, to update hidden fields on the form after the user clicks **OK**, 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 locators

A locator presents the user with a UI to locate an object in a docbase. The type of the object can be defined via component configuration or passed as a parameter by the caller component.

The following types of locators are available for specific object types:

- Sysobject (generic) locators
- User and group locators (the locators treat the group as a logical subtype of the user)
- dm_document locators
- dm_folder locators
- dm_policy locators
- dm_process 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
- flatlist

Displays all the selectable objects that meet the criteria. By default, myobject and recently used object locators display results in a flatlist.
- container

Displays a hierarchical list of objects within the selected root container. By default, subscriptions locators display results in a container (hierarchical) list.

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 [exclusionstype] and
  [attribute1] [predicate1] [value1][and | or]
  [attribute2] [predicate2] [value2] ...
```

Refer to *Web Development Kit and Webtop Reference* for a description of the configuration elements in locator components.

Adding custom help for a component

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.

Help files are mapped in the help-index component definition. The configuration file that contains this definition, help_component.xml, is located in the top 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.

When you write a new component that requires end-user help, extend the help-index component so your application can be updated without overwriting your custom help files.

The names of help files are specified in the <help-entries> element. Each filename 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>
```

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. Create a modification file in custom/config with the following skeleton content:

```
<config version="1.0"><scope>
</scope></config>
```

3. Insert entries for your help files that map each help component ID to an HTML file name. For example:

```
<component modifies="help-index:/webtop/config/help_component.xml">
  <insert path="help-entries">
    <entry id="my_component">my_component.htm</entry>
  </insert>
</component>
```

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.

Note: The help file must be in a format that can be viewed in a browser, such as HTML or PDF.

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. Your component JSP page must include this JavaScript file if the JSP page does not have a dmfw:webform tag.

To display help using a control, set the onclick attribute to call onClickHelp(). A new window is launched containing the context-appropriate help file.

If your component will be used for both portal and stand-alone web environments, name your help button DialogContainer.CONTROL_HELPBUTTON. (You must import com.documentum.web.formext.component.DialogContainer into your JSP page to do this.) Help buttons with this name will be suppressed in portal environments so portlet help can be launched instead.

Configuring Application Connector components

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

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.

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_menusgroup.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_menusgroup.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

You can remove a menu item from all Application Connectors menus with the following procedure. The following example removes the **Search** option from the menu.

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. Modify the WDK definition of this component by changing the <menugroup> element as follows:

```
<menugroup id="mymenubar" extends="appintgmenubar:  
webcomponent/config/library/appintgmenubar/apptintgmenubarmenugroup.xml">
```

3. Delete the entire <actionmenuitem> element as shown belows:

```
<actionmenuitem>  
  <aidynamic>no_dynamic</aidynamic>  
  <name>search_repositories</name>  
  <value>  
    <nlsid>MSG_SEARCH_REPOSITORIES</nlsid>  
  </value>  
  <description>  
    <nlsid>MSG_SEARCH_REPOSITORIES_DESCRIPTION</nlsid>  
  </description>  
  <imageMso>FindDialog</imageMso>  
  <component>advsearchcontainer</component>  
</actionmenuitem>
```

You can identify a menu item by finding the string in <menuitem>.<value> that matches the menu item name displayed on the menu. For strings in an <nlsid> element, match the UI string to the properties file for the component, which will give you the NLS key to find the menu item entry in the configuration file.

4. Save the custom appintgmenubar_menugroup.xml file and refresh the memory by navigating to wdk/refresh.jsp.

Modifying menu items

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 74, page 271](#) 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 74. Application Connectors menu configuration elements

Element	Description
<menugroup>	Content authoring application menu item group. The id attribute specifies the menu configuration ID. Contains one <contentsourcemenuitem>, one or more <actionmenuitem>, <nlsbundle>, and optional <menu> and <menuseparator/> elements.
<contentsourcemenuitem>	Contains one or more <value> elements
<contentsourcemenuitem>.<value>	Contains a string or <nlsid> element describing each source
<menuseparator/>	Displays a separator between menu items
<menu>	Optional. Set of <actionmenuitem> elements that form a submenu for the <menu> item. To remove an entire submenu, remove the <menu> element.
<actionmenuitem>	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 <actionmenuitem> element.
.<nlsid>	Optional key to a localized label
.<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.

Element	Description
.<component>	Maps a menu item to a WDK component. Either <action> or <component> must be present in the <actionmenuitem> element.
.<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 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.xml">
```
3. Add an <actionmenuitem> element in the desired location in the appintgmenubar_menugroup.xml file.

See [Table 74, page 271](#) 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. 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.

2. On the WDK application server machine, copy the file `webcomponent/config/library/appintgmenubar/appintgmenubar_menugroup.xml` to the WDK application `custom/config` directory.
3. Extend the WDK version of this component by changing the `<menugroup>` element as follows:


```
<menugroup id="appintgmenubar" extends="appintgmenubar:
webcomponent/config/library/appintgmenubar/apptintgmenubarmenugroup.xml">
```
4. 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, add the following filter:

```
<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>
```

5. 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>
```

6. Save your custom `appintgmenubar_menugroup.xml` file and restart the application server.

Application connector components and actions

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 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.

Application Connector components and actions are described in *Web Development Kit and Webtop Reference*. The component and action names and configuration files are prepended with *appintg*, for example, `appintgcontroller`.

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 272](#) and [Adding custom menu items, page 272](#) for more information.

- Optionally, handle or fire Application Connector browser or WDK action completion events. Refer to [Managing events, page 277](#) for more information.

The 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 appintgcontroller_component.xml) that specifies a success page:

```
<dispatchitems>
  <item>
    <name>appintgnewfromtemplate</name>
    <type>action</type>
    <successpage>opendocumentevent</successpage>
  </item>
</dispatchitems>
```

[Table 75, page 276](#) describes the configuration elements for dispatching actions or components.

Table 75. Appintgcontroller <dispatchitems> elements

Element	Description
<item>	Contains one of each: <name>, <type>, and either <page> or <successpage>
.<name>	Contains the name of a menu action, if the type is action. Contains the name of an event, if the type is page.
.<type>	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 76. Required pages in appintgcontroller component definition

Element (page name)	Description
<start>	Specifies the path to an error handler page that is loaded if the action or component is not dispatched

Element (page name)	Description
<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 Application Connectors. 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

[Table 77, page 277](#) describes controls that fire browser events:

Table 77. Application Connector control events

Control	Description
<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.

Control	Description
<dmf:fireclientevent>	Fires a client event on page rendering that is handled on the client. For events that are handled by Application Connectors, an aiEvent is fired and passed to the Application Connectors. The actual event name is passed in a <dmf:argument> element that is contained within <dmf:fireclientevent>. The Application Connectors 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 have a scope of "none".

For the first two authentication scopes, the user credentials are stored securely in the CredentialManager, a Windows executable.

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 Application Connectors 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.

To change the application server for the Application Connectors 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 of the desired instance, for example, http://localhost:8080/webtop
4. Make the desired change and restart the client application.

The user can choose a different instance of an application server content source and save it as a user preference.

In this same configuration file you can change how often the Application Connector checks for a connection to the application server. A positive value in the configuration setting results in network calls during the Office application startup and can cause a longer startup time on slower networks.

To change the application server connection check interval

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 configuration element <appIntegrationInfo>. Within that element is an attribute canConnectTimerDuration with a value in msec. The default value is 7000 (7 sec). A value of -1 specifies that the connection will never be checked, which may be preferable on a slow network.
3. Make the desired change and restart the client application.

Customizing components

Common component tasks are described in the following topics.

For information on navigating in containers and passing data to or from contained components, refer to [Configuring and customizing containers, page 304](#).

Navigating within a component

Use the method setComponentPage() in your event handler to jump to a named page in a component.

Example 6-5. 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, filename, category, and descendants information, and then navigates to the importupload page:

```
setComponentPage("importupload");
```

Jumping or nesting to another component

The method `setComponentJump()` jumps to another component. The method `setComponentNested()` nests to another component. Call `setComponentJump()` or `setComponentNested()` from within an event handler. The state of the calling component is lost when you jump to another component, but it is not lost in nesting.

If you call a component from within a dialog container, you must call `setComponentReturnJump()`, which returns to the component that called the dialog. Do not call an action from the dialog container.

The parameters for `setComponentJump()` and `setComponentNested()` are as follows:

- String name of component (required)
- String component start page (optional)
- Argument List (optional)
- Context

Example 6-6. Jumping to another component

In the following example from the `SearchEx` component class, the `navigateToFolderPath()` method calls `setComponentJump()` when the user selects a folder in the search results:

```
ArgumentList args = new ArgumentList();
if (strFolderPath != null)
{
    args.replace("folderPath", strFolderPath);
}
setComponentJump("doclist", args, getContext());
```

Example 6-7. Nesting to another component

A component can nest to another component. 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 [Adding a component listener, page 295](#).

You can add arguments to the nested form with the method `addFormNestedArgs(ArgumentList arg)`.

Example 6-8. Jumping to a node in the browser tree

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()`.

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);
    ...
}
```

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 6-9. 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 6-10. 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);
```

Passing arguments from a control to a component

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 6-11. Passing arguments to a component

The following example in `saveCredential.jsp` sets the value for the repository name using the argument tag. This passes the repository (docbase) name to the `onClickRemove` event handler.

```
<dmf:link nlsid="MSG_REMOVE" onclick="onClickRemove">
  <dmfx:argument name='docbase' datafield='docbase' />
</dmf:link>
```

Note: If you are going to use the argument in your component class, name it as a parameter in your component definition. If you are going to pass it along to another component, as in the example, add it to an `ArgumentList` object that you pass to the component, as in the following:

```
public void onClickRemove(Control control, ArgumentList args)
{
    args.add(ATTR_COMPONENT, COMPONENT_REMOVE);
    setComponentNested(COMPONENT_CONTAINER, args, getContext(), this);
}
```

Passing arguments between components

Example 6-12. 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 6-13. 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();
```

Accessing an included component

Example 6-14. 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 then access and set values in the included component by getting the `componentinclude` control in your component class.

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 component through the `componentinclude` control as in the following example:

```
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 6-15. 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();
```

Implementing a component

All content components should must implement the following lifecycle methods and functionality. Other methods are optional as noted.

- `onInit()`

Should accept appropriate arguments and supply valid defaults for non-mandatory parameters. Initialize controls to valid initial defaults. Set connections on databound controls. Read standard and custom component definition settings such as object type filter values, object types to display, and visible columns. Call the superclass `onInit()` before you add your custom operations:

```
public void onInit(ArgumentList args)
{
    super.onInit(args);
    ...
}
```

You can read preferences or perform other component functions that do not require user input.

- `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.

If your component needs to get data, you can get a data provider in the `onRender()` method after getting the DFC session. 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());
}
```

- `onRefreshData()`

Override the Component class method `onRefreshData()`. The framework calls this method after the execution of an action that has invalidated the data being displayed by this component.

- `refresh()`

Call `refresh()` on all data controls in the component to automatically re-query the data or to rebuild custom `ScrollableResultSet` objects. For an example, refer to [Refreshing data, page 207](#).

- readConfig()

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);
    }
    ...
}
```

Additional features that are commonly implemented by components are the following:

Object type filters — If your component displays a list of Documentum objects, the component should provide a drop-down 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.

Title — All content components should provide a title to indicate the component name. Additionally, content components that display Documentum objects should display the current repository and username in the format "MyComponent (repository : 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.

Rendering messages to users

The message service maintains in the session a list of messages to the user from various services and components. 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:

```
//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 each application can have its own error message service.

For information on using tracing to track down errors, refer to [Tracing, page 446](#).

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:

```
catch (Exception e)
{
    throw new WrapperRuntimeException(e);
}
```

Reporting user errors — 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 Java EE server that it is an error page. The servlet is called by the Java EE 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.

Supporting export to CSV

The following Webtop components support the `exporttocsv` action, which exports the content of a datagrid to a CSV file: `home cabinet classic`, `cabinets (objectlist)`, `search results`, and `my files`. In other components, the `exporttocsv` action will be disabled. To add support for export to CSV, your component must implement the `ITableDataProvider` interface. Your component must pass the correct, formatted information. In general, to pass data to be formatted to CSV, the component should obtain the datagrid control, iterate through the datagrid rows and visible columns to obtain the datagrid field values, apply any value formatters, and pass this data through the interface.

Note: If the data for export to CSV contains double-byte characters, the CSV file cannot be opened in Microsoft Excel without corruption. Excel has a bug reading unicode characters that are encoded in UTF-8 unless they are imported through the Excel import feature. The user must save the file on the local host and import it into Excel using the Data Import External Data menu.

Implementing `isDataAvailable()` — Returns whether data is retrievable.

```
public boolean isDataAvailable()
{
    return true;
}
```

Implementing `getMetadata()` — The following example from `HomeCabinetClassicView` gets the metadata for export to CSV. Retrieves the number, types and properties of datagrid columns.

```
public Metadata getMetadata()
{
    ComponentColumnDescriptorList compColumnDescList =
        new ComponentColumnDescriptorList(this, "columns", getColumnPreferenceId(
        ));
    List columnDescList = compColumnDescList.getColumnDescriptors();
    return TableDataProviderUtil.getMetadata(columnDescList);
}
```

Implementing `rowIterator()` — This method returns an iterator over the rows. The order of the returned rows cannot be predicted in this implementation.

```
public Iterator<Row> rowIterator(List<String> columnNames)
```

```

{
  Datagrid datagrid = (Datagrid)getControl(CONTROL_GRID);
  return TableDataProviderUtil.getRows( datagrid.getDataProvider(), columnNames);
}

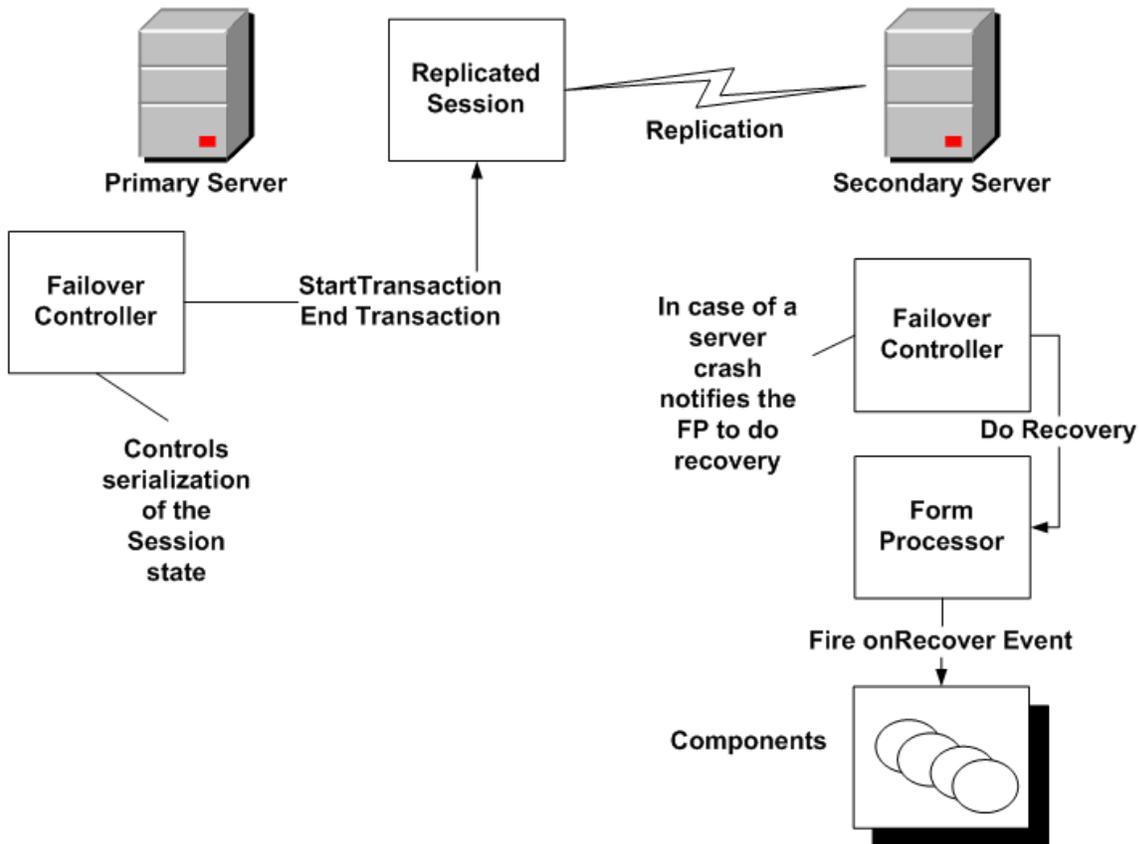
```

Implementing failover support

The topic [Enabling application failover support, page 86](#) 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. [Figure 15, page 292](#) diagrams this process.

Figure 15. 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-based application deployment guide for instructions on how to configure 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 datatype 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. (Use `ClientSessionState` if you have customized the application to launch more than one session for a user.) 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: preferredrenditionservice,
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 6-16. 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();
        ...
    }
}

```

Adding 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 6-17. 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 6-18. 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", "aclocator");
    selectArgs.add("flatlist", "true");
    setComponentNested(
        "aclocatorcontainer", 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 6-19. 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;...}}

```

Supporting clipboard operations

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 username and password).

The following topics describe the implementation of clipboard support in a custom component.

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. For examples of implementations of the clipboard handler methods, refer to the source code provided in `webcomponent/src/com/document/webcomponent`.

Getting the user location for clipboard operations — 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. You can use `FolderUtil` utility class `com.documentum.web.formext.docbase.FolderUtil` to get the folder ID:

```
m_strFolderId = FolderUtil.getFolderId(strFolderPath);
```

After a clipboard operation, the component that launched the clipboard may need to refresh the display to show changes from the operation. Override `onRefreshData()` and call `refresh()` on any data provider controls such as data grid. This will reload the data that may have changed.

Supporting drag and drop

Refer to [Configuring drag and drop, page 98](#) and [Configuring the Active-X plugin install, page 97](#) for information on enabling and configuring drag and drop support in the application.

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.

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 78. Configuration elements (<dragdrop>)

Element Name	Description
<code><sourceactions></code>	Contains zero or more <code><sourceaction></code> classes that support drag actions on sources in the component. This element must be declared even if it contains no <code><sourceaction></code> elements, in order to enable the component as a drag source.
<code><sourceaction></code>	Contains a fully qualified class name of class that implements <code>IDragSourceAction</code>
<code><targetactions></code>	Contains zero or more <code><targetaction></code> classes that support drop actions on drop targets in the component
<code><targetaction></code>	Contains a fully qualified class name of class that implements <code>IDragTargetAction</code> , for example, <code>com.documentum.web.formext.control.dragdrop.CopyToFolderTargetAction</code> . To remove a drop action, remove this element from definition.

Element Name	Description
<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 from or drop to 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. 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 so a tooltip can be displayed.

Dragging — Most objects that a user can add to the clipboard are potential drag sources. 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, virtual documents (in the default rendition), folders or folder subtypes such as cabinets or rooms, documents to be checked in, and documents for which to add a rendition. The following objects cannot serve as targets: windows or frames in a different session, repository node in the navigation tree, inbox, categories, administration, and the root cabinet.

Drop actions supported by the component should be listed in the <dragdrop>.<targetactions> element of the component definition. [Table 79, page 300](#) describes the target action classes that are available in WDK, in the package com.documentum.web.formext.control.dragdrop.

Table 79. WDK drop target actions

Action class	Action displayed
CopyToFolderTargetAction	Copy
MoveToFolderTargetAction	Move (default action if listed in the configuration)
LinkToFolderTargetAction	Link
AddVirtualDocumentNodeTargetAction	Add
RepositionVirtualDocumentNodeTargetAction	Reposition
SubscribeAction	Subscribe (default action if listed in the configuration)
ImportTargetAction	Import (accepts files from desktop; is default action when target is a folder or virtual document)
CheckinFromFileTargetAction	Checks in a file dropped onto a file that is checked out by the same user
AddRenditionTargetAction	Imports a rendition that is dropped onto a file

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 IDragTargetAction instance for each action to the IDragTarget 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 .0dnd 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 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>

```

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();
}%>

```

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);
    ...}
```

To make a component within a container 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 a properties. a pop-up window displays trace output as in the following excerpt:

```
modal.js: Resizing Frame : timeoutcontrol
modal.js: Resizing Frame : titlebar
...
```

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.

Configuring and customizing containers

The following topics describe configuring and customizing containers.

Calling a container by URL, JavaScript, or action

Containers can be invoked in the following ways:

- By URL
- By JavaScript
- By an action definition
- By a server class

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.

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);
```

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.

Requiring a component visit within a container

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`.

Example 6-20. 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. In this case, the component must be contained within a container that extends `propertySheetContainer`. The example would be as follows in the attributes configuration file:

```
<requiresVisitBeforeCommit>true</requiresVisitBeforeCommit>
```

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 435 for more information on context)
- `returnListener`: An implementation of `IReturnListener`, with `setComponentNested()` only. Refer to [Adding a component listener](#), page 295 for more information.

Example 6-21. 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. 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**). In the following example from `AdminDelete`, the `onCommitChanges()` method performs the commit:

```

public boolean onCommitChanges ()
{
    destroyObject(m_objectId);
    return true;
}

```

- `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`. 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;
}
```

Note: If both `canCommitChanges()` and `canCancelChanges()` return `false`, a `Close` button is displayed in place of the `OK` and `Cancel` buttons.

- `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`. 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 `requiresVisitForCommit` configuration attribute value in the container definition. Refer to [Requiring a component visit within a container, page 306](#) 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

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 [Requiring a component visit within a container, page 306](#) for more information.

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. To access a non-current component or multiple components in the container, iterate through the array that is returned by `getContainedComponents`.

Example 6-22. Initializing contained components

You can initialize all of the contained components using `setCurrentComponent()` and `initContainedComponent()`. The following example from `TaskMgrContainer` does initialize 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 6-23. Getting the current component

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 6-24. 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
    }
}
```

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. The **Next** and **Previous** buttons are hidden if both `hasNextPage()` and `hasPrevPage()` return false.

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 6-25. 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 6-26. 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
}
}
```

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 6-27. 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.

Using combocontainer

The combocontainer can be used for actions on multiple objects. Specify combocontainer or a container that extends it for your action definition for multiselect actions (actions called by an action tag with the dynamic attribute value of multiselect).

Note: You cannot call a combocontainer or container that extends it by URL. Instead, you must call it by an action that uses the LaunchComponent execution class or class that extends LaunchComponent. This execution prepares the arguments required by the combocontainer class.

Propagating control values to contained components — The combocontainer propagates modified control values to the next component in the container. For example, when the user checks in multiple documents, the values from the first checkin component instance are propagated to the UI of the second checkin instance. The enabled and visible state of the control is not propagated, but you can configure the propagation of those values in the component definition file. Add a <control-propagation> element to the component definition and one or more <property> child elements, with the following syntax:

```
<control-propagation>
  <property name="enabled"/>
  <property name="visible"/>
</control-propagation>
```

By default, multiple selection arguments are passed by the action execution class in a URL to the container and then in a URL to each component in the container. You can configure your action to cache multiple selection arguments to enhance performance. For information on caching, refer to [Caching component arguments for multiple selection, page 159](#).

Using a prompt (pop-up) 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 you can call setComponentNested().

The following procedure uses an example that adds a prompt pop-up to a container event handler. It uses the newdoccontainer component and adds a confirmation dialog when the user clicks the **OK** button to create a new document. You can modify this to test for certain conditions before execution, such as the selected object type. You can also pop up the prompt from other types of containers.

To add a pop-up prompt to a container

1. Modify the newdoccontainer component definition as follows:

```
<component modifies="
  newdoccontainer:webcomponent/config/library/create/newdoccontainer_
  component.xml">
  <replace path="class">
```

```

    <class>com.mycompany.CustomNewDocContainer</class>
  </replace>
  <replace path="nlsbundle">
    <nlsbundle>
      com.documentum.webcomponent.library.create.NewContainerNlsProp
    </nlsbundle>
  </replace>
</component>

```

2. Create the properties file referenced in the <nlsbundle> element as a text file named CustomNewContainerNlsProp.properties in WEB-INF/classes/com/mycompany, with the following content:

```

NLS_INCLUDES=com.documentum.webcomponent.library.propertywizardcontainer.
PropertySheetWizardContainerNlsProp
MSG_WINDOWS_TITLE=New from Template
MSG_HEADING_TITLE=Enter new document name and properties

#new content for custom newdoccontainer
MSG_CONFIRM_DOC_TITLE=Confirm document creation
MSG_CONFIRM_DOC_MESSAGE=OK to create new document?
#end new content

MSG_OK=Finish
MSG_CREATE=Create
MSG_CREATE_TEMPLATE_TIP=Create Template

#Accessibility strings
MSG_OK_TIP=Finish
MSG_VALIDATION_ERROR=There are validation errors -- {0}

```

3. Create the Java class CustomNewDocContainer in WEB-INF/classes/com/mycompany with the following content:

```

package com.mycompany;

import com.documentum.web.formext.component.Prompt;
import com.documentum.web.form.control.databound.DataDropDownList;
import com.documentum.web.common.ArgumentList;
import com.documentum.web.form.FormActionReturnListener;
import com.documentum.web.form.Form;
import java.util.*;

public class CustomNewDocContainer extends
    com.documentum.webcomponent.library.create.NewDocContainer{

    public void onOk(com.documentum.web.form.Control button,
        com.documentum.web.common.ArgumentList args)
    {
        //Read your custom strings from NLS properties file
        args.add(Prompt.ARG_TITLE, getString("MSG_CONFIRM_DOC_TITLE"));
        args.add(Prompt.ARG_MESSAGE, getString("MSG_CONFIRM_DOC_MESSAGE"));
        args.add(Prompt.ARG_BUTTON, new String[]{Prompt.YES, Prompt.NO});

        setComponentNested("prompt", args, getContext(
            ), new FormActionReturnListener(
                this, "onReturnFromPromptInput"));
    }
}

```

```
}  
  
public void onReturnFromPromptInput(Form form, Map map)  
{  
    // check whether the confirmation prompt has returned  
    String strButton = (String)map.get(Prompt.RTN_BUTTON);  
    if (strButton != null && strButton.equals(Prompt.YES) )  
    {  
        //If confirmed, allow super event handler to execute  
        //If not, do nothing  
        super.onOk(null, null);  
    }  
}  
}
```

4. Compile your class and restart the application server. When you create a new document and click **OK**, you should see the following dialog:

Figure 16. Pop-up prompt



Creating modal containers

Any component can be modal. (Refer to [Using modal windows, page 303](#) 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 that have 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.)

Containers overview

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 include one component within another using the `<componentinclude>` tag.

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 do the following:

- Require a visit to one or more components before changes are committed.
- Change container labels using the string properties file for the container.
- Set the default component by listing it first within the `<contains>` element.

The container component definition includes a component parameter. If you call a container with this parameter, it opens the container with the specified component in focus. The container start page is defined in the container configuration file as the value of the `<pages>.<start>` element.

A container has the following characteristics:

- The container class extends `com.documentum.web.formext.component.Container`
- 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 components in the container. 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 separate instance of the Checkin component).
- A container can display only one component at a time.
- The currently displayed component is accessible through methods on the Component class.
- 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.

Some components cannot stand alone and must run within a container. For example, components within a container that extends the `combocontainer` cannot be called directly. As a general rule, if the component is called by an action that launches its container, you cannot call that component directly. You can call these components and their container by an action that uses the `LaunchComponent` action class.

The content transfer components (`cancelcheckout`, `checkin`, `checkout`, `edit`, `export`, `import`, and `view`) need access to content transfer configuration in the container definition. Therefore, these components must be called within their containers. The `newdocument`, `newfolder`, and `newcabinet` components must also be used within their respective containers.

For information on calling a container by an action or URL, refer to [Calling a container by URL, JavaScript, or action, page 304](#).

Choosing a container

Use or extend a WDK container that is appropriate for your navigation purposes. [Table 80, page 317](#) describes the WDK container types and their features.

Table 80. Conditions for selecting WDK containers

Condition	Container name
Extended by <code>dialogcontainer</code> , no layout	<code>abstractcontainer</code>
Single component Buttons OK, Cancel, Close, Help	<code>dialogcontainer</code>
Single navigation component breadcrumb control	<code>navigationcontainer</code>
Single component with ordered pages (Previous , Next)	<code>wizardcontainer</code>
Multiple selection	<code>combocontainer</code>
Multiple contained components	<code>propertysheetcontainer</code>
Multiple contained components with wizard navigation	<code>propertysheetwizardcontainer</code>
Content transfer	<code>contentxfercontainer</code>

Using Business Objects in WDK

Business objects 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 calls several business objects, including inbox, workflow reporting, subscriptions, and content intelligence services. Business objects can be installed on the application server and called from a WDK-based application. For information on how to create business objects, refer to the *Documentum Foundation Classes Development Guide*.

Note: You can install business objects on the Content Server as server methods, jobs and lifecycle procedures, but they cannot be called from a WDK-based application.

A type-based business object (TBO) allows you to override DFC methods for a specific object type. 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`. 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.

A service-based object (SBO) method can be called from any WDK action or component class. You must instantiate the SBO to call its methods in your application.

To call an SBO method in WDK

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 of the SBO.

Example 6-28. Calling an SBO method

The following example from `EditTaskNote` instantiates the inbox service and calls a service method:

```
protected ITask getTaskObject(String taskId) throws DfException
{
    // manufacture task using an instance of inbox service
    IDfSessionManager manager = SessionManagerHttpBinding.getSessionManager();
    IInbox inboxService = (IInbox)DfClient.getLocalClient().newService(
        IInbox.class.getName(), manager);
    inboxService.setDocbase(getCurrentDocbase());

    return inboxService.getTask(new DfId(taskId), true);
}
```

Components overview

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.

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 and Webtop Reference*.

The following topics describe components in detail.

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). .

- 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.

- 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 runtime (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 definition (XML)

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_HOMEwdk/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">
   </param>
 </params>

5 <pages>
   <start>/custom/.../some_page.jsp</start>
   <custom_page_namecustom_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 40](#). The component version support is described in [Versioning a component, page 250](#).

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 280](#).

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 419](#).

8 Contains control initialization settings for controls in the component JSP pages. For more information on these classes, refer to [Initializing content transfer controls, page 412](#).

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 a user with the role of administrator is permitted to see all attributes of an object.

Component JSP pages

A JSP file contains layout for components. The layout is made up of HTML, JavaScript, and Documentum JSP tags.

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.

Forms are not configurable. Elements in the form (JSP and HTML tags) are configurable. Components that contain the form are configurable in an XML component definition.

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 reference by `<%@ include file='doclist_thumbnail_body.jsp' %>` is in the same directory as the including file. The included file reference by `<%@ include file='/wdk/container/modalDatagridContainerStart.jsp' %>` is in the `wdk` subdirectory of the web application.

Component tag classes

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.

If your component does not need to support behavior in addition to that provided by `com.documentum.web.formext.component.Component`, use the `Component` class for your custom component class in the component definition.

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()`, 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.

The `FormTag` class (`dmf:form`) generates the HTML for a form. This tag generates hidden input fields and JavaScript for scrolling, sets modality, and reloads the URL if the `keepfresh` attribute is set to `true`.

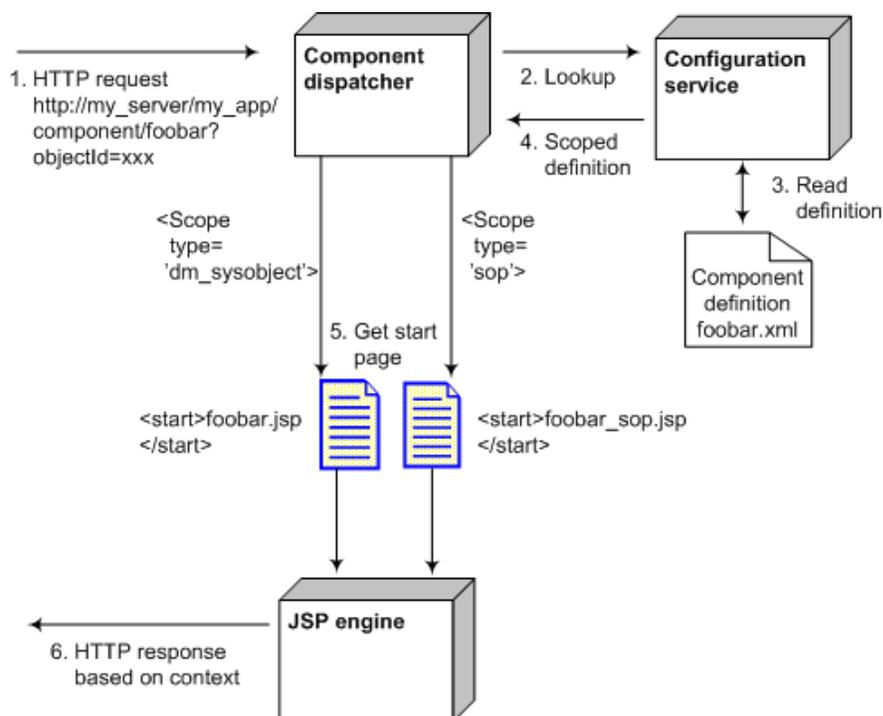
The `WebformTag` class (`dmf:webform>`) renders a list of script and style sheet inclusions whose content is driven from the resource bundles `com.documentum.web.form.WebformScripts` and `com.documentum.web.form.WebformIncludes`.

Component processing

Component JSP pages are processed and validated by the form processor. For information on control validation, refer to [Validating a control value, page 156](#).

Figure 17, page 325 shows the sequence of processing that occurs when a component is called via URL.

Figure 17. Component processing sequence

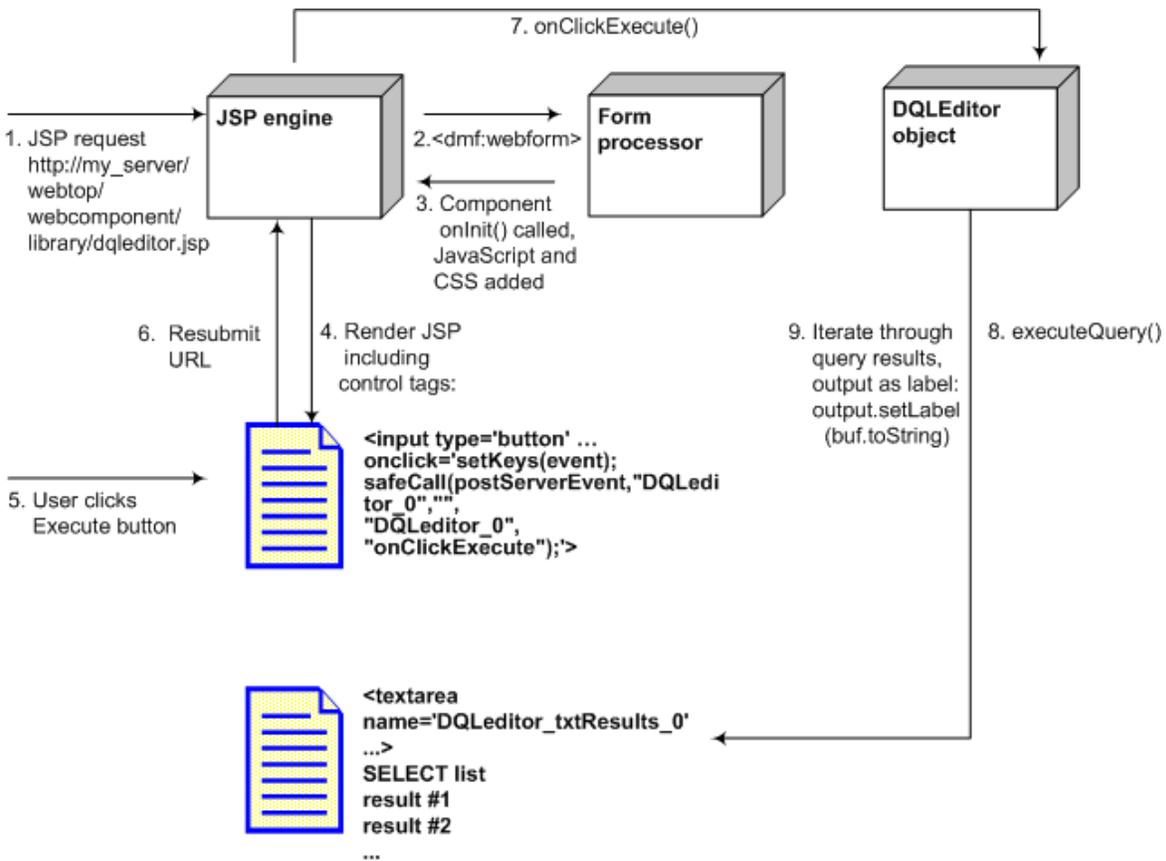


In the figure, a typical component is processed in the following sequence:

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.

Figure 18, page 326 shows the interaction between a component JSP page, user, and UI controls.

Figure 18. JSP page, control, and user interaction



In the figure, 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()`.

Configuring and Customizing Multi-Component Features

This chapter describes how to configure and customize features that affect several parts of the application.

- [Configuring and customizing preferences, page 329](#)
- [Using custom presets, page 343](#)
- [Configuring and customizing strings, page 346](#)
- [Branding an application, page 356](#)
- [Creating asynchronous jobs, page 364](#)
- [Customizing authentication and sessions, page 369](#)

Configuring and customizing preferences

The following topics describe tasks that configure and customize component-level and user preferences.

Preferences and cookies

User preferences can be session-limited or cookie-based. All cookie-based preferences are stored for each user in a preference repository that is configured in app.xml. Any cookie that should not be stored must be specified in app.xml. For example, login cookies are used before the user's preferences are downloaded from the preference repository, so they are specified as non-repository preferences. For information on configuring a preferences repository and non-stored preferences, refer to [Configuring a preference repository and preference cookies, page 83](#).

Autocompletion lists for a user are stored in the repository only and not as a cookie.

When a user logs in, the user's preferences are downloaded and cached on the application server. Preferences are stored for the first username that the user uses to log into a repository. For example, if the user logs into the WDK-based application with the user ID of kasanr and then logs into another repository as the user testuser, the user preferences are stored under the ID kasanr. The object in the preference repository will be kasanr.preferences. The preferences are stored as name/value pairs.

During the user's session, updates to preferences are written to the Webtop cookie and the in-memory cache of the preference repository store. When the user logs out or the session times out, changed preferences are written to the preference repository.

Cookie information is optimized to store preference IDs instead of names. The properties file `CookieJarProp.properties` in `WEB-INF/classes/com/documentum/web/common` contains the mapping of preferences to ID. Add an entry to this file when you create a new preference.

Configuring user 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 81. User preference components

Webtop Label	Component ID	Description
General	general_preferences	View, entry component, theme, accessibility, checkout location, drag and drop
Columns	display_preferences	Attribute columns to be displayed for each configured component. Refer to Configuring user column display preferences, page 331 .
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

You can also set hidden debug preferences for a development environment by calling the `debug_preferences` component by URL. For example:

```
http://localhost/webtop/component/debug_preferences
```

Configuring user column display preferences

User display preference default 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 ([Configuring a component-level preference, page 337](#)). Each component can have an entry that gives the user the ability to select columns or other preference settings.

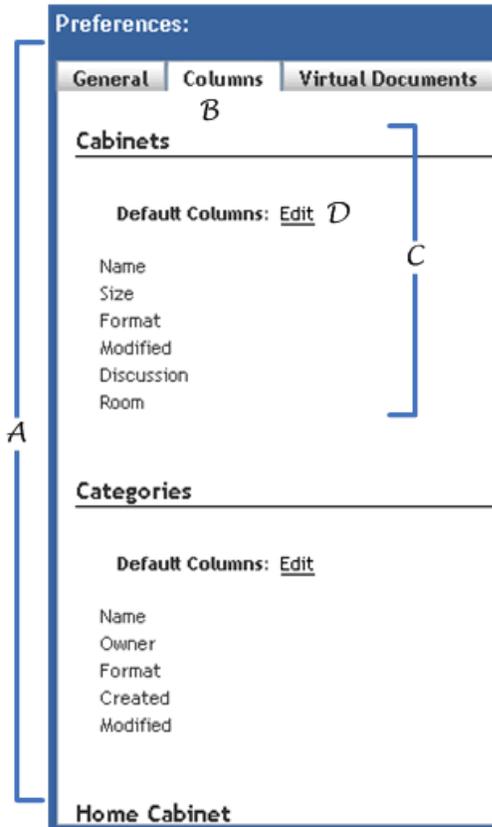
The preferences elements specific to the `display_preferences` component are as follows:

Table 82. Column display preference elements

<code><preference></code>	ID for the column display preference
<code><display_docbase_types></code>	Sets the contents of the drop-down list to specify column preferences for an object type. Contains <code><docbase_type></code> elements. You can add this element to any <code><preference></code> element in the <code>display_preferences</code> component to override the default object type list.
<code>.<docbase_type></code>	Adds a repository type to display in the drop-down list of types for setting column display preferences. Contains a <code><value></code> element whose value must correspond to a type in the data dictionary and a <code><label></code> element that will display a label for the type.
<code><show_repeating_attributes></code>	Set to false to not display repeating attributes. Will not affect attributes in the default list or attributes already in the selected list.
<code><enableordering></code>	Generates up and down arrows that allow the user to reorder columns. Default = true

The following diagram illustrates the UI of the column `display_preferences` component and how it is generated. Note that the columns that are available for user preferences can be limited by a preset.

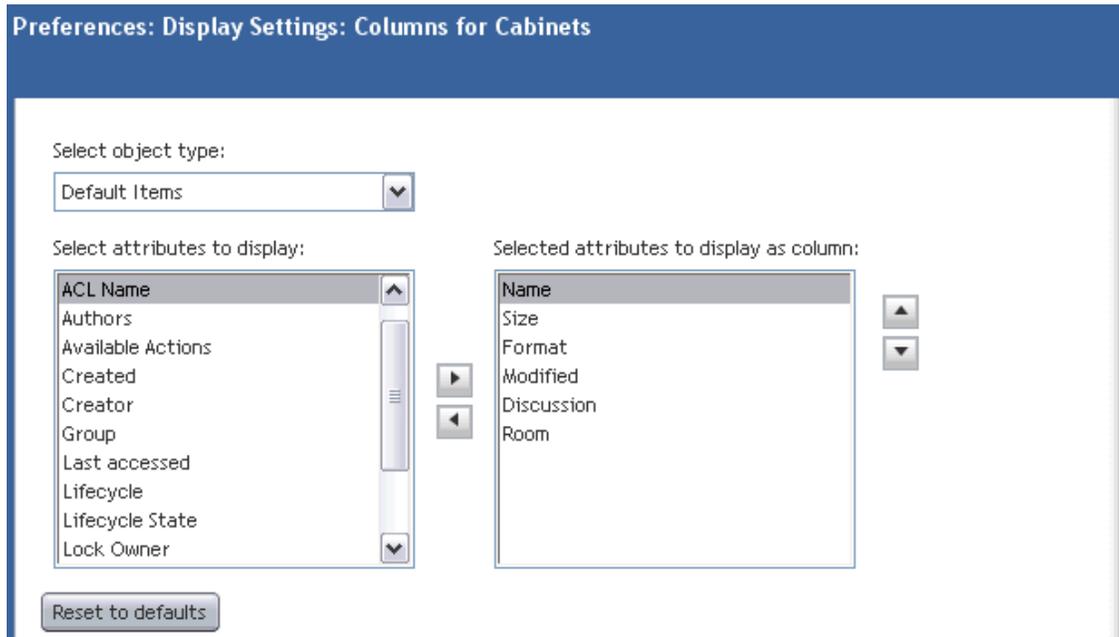
Figure 19. WDK display preferences



- A: Preferences container UI with header title and footer buttons. Specified by preferences_component.xml, the container JSP page is provided by the parent property-sheet-container component.
- B: Hidden preferencescope tag in display_preferences.jsp, which sets the scope to 'User'
- C: Generated by <preference> element in this (display_preferences) component definition. The first <preference> element id is application.display.classic_cabinets_columns, and the <value> element specifies that the column list comes from the definition for the component[id=doclist].columns. In the doclist component definition, the <columns> element specifies the visible columns. This list can be overridden by a preset.

- D: Button that launches the component named in the <editcomponent> element, in the container named in the <editcontainer> element. If an <editcomponent> element is present with a valid value, this button is generated. In the example shown below, the columnselector component is launched by the **Edit** button to edit the columns preference. The attributes are generated by an attributeselector control.

Figure 20. Column selector component



The column selector component UI is generated by the <preference> element in the display_preferences component definition (display_preferences_ex_component.xml in webtop/config). The above example is generated by the preference element with the id application.display.classic_cabinets_columns. The <value> element within this preference for classic cabinets inherits the columns configured in component[id=doclist].columns:

```
<preference id="application.display.classic_cabinets_columns">
  ...
  <value>component[id=doclist].columns</value>
</preference>
```

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.

The following procedure adds a custom type to the display_preferences component definition so the user can configure display of custom attributes.

To add a display preference for a custom type

1. Create a modification file in custom/config with the following skeleton content:

```
<config version="1.0"><scope>
</scope></config>
```

2. Modify the component definition, for example:

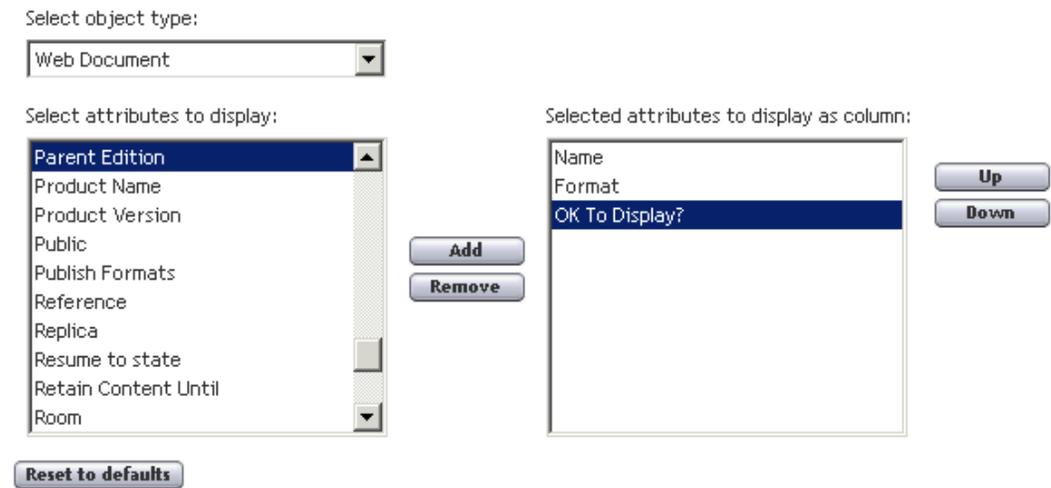
```
<component modifies="
  display_preferences:webcomponent/config/environment/
  preferences/display/display_preferences_component.xml">
```

3. Add your custom type to the <display_dochbase_types> element, similar to the following:

```
<insert path="display_dochbase_types">
  <dochbase_type>
    <value>technical_publications_web</value>
    <label>Web Document</label>
  </dochbase_type>
</insert></component>
```

4. 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 21. Custom type column preferences



Customizing user preferences

The WDK preferences container 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 component class stores preferences using the PreferenceService class.

User preferences are stored as in the global repository by default. Preferences are stored by application, application version, and user. Preferences are cached on the application server during the user's session.

WDK 5 stored preferences as cookies. These cookies are read and saved to the global repository in WDK 6 on the user's first login.

A small subset of preferences are cookie-based. You can add other cookie-based preferences and register them in app.xml. For information on specifying cookie-based preferences, refer to [Configuring a preference repository and preference cookies, page 83](#).

Storing user preferences — `IPreferenceStore` provides an interface to the `PreferenceService` storage mechanism with methods for reading and writing preference values. The actual storage mechanism can consist of a file system, repository, or cookies. Only one instance of the preference store is used for each HTTP session.

Get the preference store by importing `IPreferenceStore` and `PreferenceService` into your action or component class and then calling `getPreferenceStore()`:

```
import com.documentum.web.formext.config.IPreferenceStore;
import com.documentum.web.formext.config.PreferenceService;
...
IPreferenceStore preferenceStore = PreferenceService.getPreferenceStore();
```

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: Minimize the number of preferences stored as cookies. Since cookies are passed back and forth with every request and response, there is a small increase in network traffic. To minimize network traffic or service users on low-bandwidth connections, do not use preferences. Additionally, cookies are stored on the browser's local machine, so preferences for roaming users is not supported.

Example 7-1. 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 username 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 7-2. Retrieving user preferences

By default, the read methods such as `readBoolean` or `readString` 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 username preference:

```
//Get a username preference value
IPreferenceStore store = PreferenceService.getPreferenceStore();
String strUsername = store.readString ("Username");
```

Preferences storage model — Preferences are stored in the global repository in the following folder structure:

```
/Global Repository Name
  /Resources
    /Preferences
      /app name
        /app version
          /user_name.xml
```

The user-specific XML preferences file has the following syntax:

```
<prefs>
  <pref>
    <name>preference_name</name>
    <value>preference_value</value>
  </pref>
</prefs>
```

Bypassing user preferences

The configuration lookup methods `lookupString`, `lookupInteger`, and `lookupBoolean` have an optional parameter `consultPreference`. Set to `false` to look up a configuration value from the component definition and bypass a lookup of the user preference when the lookup is not needed.

Configuring a component-level preference

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 login component definition in `webtop/config` defines a preference of centering the login dialog through a custom element, `<centered>`. The value of this element is used to set the preference for all users.

User preferences within a component are configured through a `<preferences>` element. [Table 83, page 337](#) shows elements that can be used within a `<preferences>` element to define preferences for a component:

Table 83. <preference> elements

Element	Description
<code><preference id='name' disabled='true'></code>	Defines the preference and its required ID attribute. Disabled attribute is optional and defaults to <code>false</code> .
<code><label></code>	Required. Sets the display name of the preference.
<code><description></code>	(Optional) Specifies a description to be displayed beneath the main row of subcontrols
<code><nlsid></code>	(Optional) Specifies an NLS key that is resolved to a string in the <code><nlsbundle></code> class referenced in the component definition file. Use this element inside <code><label></code> and <code><description></code> .
<code><type></code>	Required. Specifies the type of preference value. Valid types: <code>int</code> <code>string</code> <code>boolean</code> <code>columnlist</code>
<code><value></code>	Specifies the default value for the preference
<code><position></code>	Specifies the position of the value control: <code>right</code> (default) <code>left</code>

Element	Description
<display_hint>	Forces the control to use a type of display. Valid values: password (for strings or integers) dropdownlist listbox hidden. A <constraints> element containing one or more <element> elements must be present for drop-down list or list box.
<listbox_size>	Specifies number of visible elements for a list box (default = 5). Use this if the value of <display_hint> is listbox.
<constraints>	Specifies preference constraints. If the <display_hint> element is absent, the preference control will look for a <lowerend> and <upperend> for an int type. For other types this element will be ignored. With the <display_hint> element present, the control will look for <element> instead of <constraints>.
<lowerend>	Specifies the lower limit for int. For other types, and when the display_hint element is present, <lowerend> will be ignored.
<upperend>	Specifies the upper limit for int. For other types, and when the display_hint element is present, <upperend> will be ignored.
<element>	Specifies an element in a list of int or string type. Use with <display_hint> to build a list of elements for a drop-down list or list box. The attributes on <element> specify both the preference value and its description.
<element value= <i>some_value</i>	Displays a description that is different from the preference value. The element content provides the description, and the value attribute provides the preference value.
<editcomponent>	Specifies the component to launch for editing the preference. Configures column lists only (<type> columnlist</type>).

Element	Description
<editcontainer>	Specifies the container to launch for editing the preference. Configures column lists only (<type> columnlist</type>).
<inherits>	Specifies the preference definition that sets the columns to be displayed. Configures column lists only (<type> columnlist</type>).

String and integer preferences are displayed in a text box (default), list box, or drop-down list. 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.

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-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>

```

Using custom presets

Presets provide a way for users to configure a WDK-based application through a presets editor UI. (Configuration is described in [Chapter 1, Configuring WDK Applications](#).) Refer to the *Webtop User Guide* for information on using the presets editor.

The following topics describe presets and how to use them in a custom component. Adding filters to the presets editor is not supported.

About presets

To give users the ability to create presets using the presets editor, assign those users the role `dmc_wdk_presets_coordinator`. Users who are authorized to create presets (preset designers) will create sets of criteria that govern the UI that is displayed to a user based on the context. The context reflects the current state of an application, including user role, current application name, object type, current lifecycle state; current repository; and custom contexts.

Presets can be disabled by adding the elements `<presets><enabled>>false</enabled></presets>` to `custom/app.xml`. By default, they are enabled for Content Server version 6 repositories. You can specify the presets repository in `custom/app.xml`. If this repository is not configured, the global repository is used. For more information on the configuration in `app.xml`, refer to [Configuring presets access, page 82](#).

The BOF2 preset service enables the presets editor. In the presets editor, preset designers create values for a preset item in the desired scope. For example, a preset designer could create one set of formats to be displayed to contributors and another, larger set to administrators. In this example, the scope is role and the preset filter is formats.

Presets support the following scopes: roles, users, groups, location, and types. The preset can include a filter that displays a subset of available objects. Filters can be applied to actions, action exclusions,

locations, object types, formats, document or workflow templates, lifecycles, retention policies, saved searches, permission sets, groups, columns for display, and autoattributes. Presets are defined in the presets editor and stored the presets in a presets repository.



Caution: Do not modify presets by editing the preset definitions stored in the repository. Modify them using the presets editor.

The WDK DocApp creates a presets folder, owner, and permission set. The presets folder is `/Resources/Registry/Presets/Webtop`. The owner of the presets folder is the user `dmc_wdk_presets_owner`. The folder is assigned the permission set ACL `dmc_wdk_presets_acl`.

Stored presets can be shared between applications and can be modified after deployment. The preset items should be applicable to all applications that share presets. Each application that shares presets should specify the same presets folder and repository in `app.xml`.

When a WDK-based application starts up, presets are downloaded from the presets repository to the application server. Presets are checked for changes at a configurable interval. This interval is set as the value of `<presets>.<refresh_period_seconds>` in `custom/app.xml`. If the presets repository is not online when the WDK application starts up, or if the user context is not governed by any existing presets, the configurations that are defined in the component and action configuration files are used. Presets override WDK configurations by dynamic insertion of filters in memory. The configuration files on the application server are not changed.

The presets cache is located in a Documentum directory under the application server binaries directory. Under the presets cache directory is a subdirectory for each Web application. The name of the presets cache file is created from the `r_object_id` of the `dmc_preset_package` and its saved `i_vstamp` number.

Using presets in a component

Add a skeleton preset definition to your component definition to specify the configuration element that should be user-configurable through the presets editor. For example, the checkin component definition adds a format filter item to the definition as follows:

```
<preset_item id="format_type_filter">
  <selection_filter>
    <filter_items>
      <item></item>
    </filter_items>
  </selection_filter>
</preset_item>
```

If the user's context matches the scope of a preset (custom setting) that contains the format filter, then the user will see only the available formats for that preset definition. For example, a preset may be defined for the contributor role that contains a short list of five formats. On checkin, a contributor will be able to select only from the list of five formats. An administrator sees the full list of formats in the repository unless there is another preset list of formats for administrators.

Selector filters — The `<selection_filter>` element in the component definition is resolved by the WDK configuration service to a filter class that extends `com.documentum.web.preset.Selector`. Available filter classes in the package `com.documentum.web.preset` include `FormatSelector`, `TemplateSelector`, and `TypeSelector`. Extend the generic `Selector` class to filter any general list of items.

Locator filter — The `LocatorFilter` class is used by locator components to provide preset filtering of objects that users can select in the locator. The syntax in the locator component definition is similar to that for other selectors. Support for `LocatorFilter` is built into `LocatorQuery` and `ObjectLocator` classes. To add a locator filter to your custom locator, add a `<preset_item>` element. The following example adds a group locator filter:

```
<preset_item id="group_locator_filter">
  <selection_filter>
    <filter_items>
      <item></item>
    </filter_items>
  </selection_filter>
</preset_item>
```

In the component class, use the filter class to filter the list that will be presented to the user. For example, the `Checkin` class applies the format selector as follows. Note that the value of the `id` attribute of the `<preset_item>` element in the component definition is passed to the `FormatSelector` class.

```
protected ScrollableResultSet createFormatsResultSet()
{
    ...
    TableResultSet tmpResultSet = new TableResultSet(
        new String[]{"name", "description"});
    FormatSelector formatSelector = new FormatSelector(
        getConfigBasePath(), FORMAT_PRESET_ID, getContext());
    List<Item> filterList = formatSelector.getSelection(dfSession);
    for (Item item : filterList)
    {
        //create result set
    }
}
private static final String FORMAT_PRESET_ID = "format_type_filter";
```

Looking up preset data within a component

Use the configuration service lookup within WDK-based components to look up preset values. The following example looks up the `<preset_item id=entry_section>` in the component definition:

```
<preset_item id='entry_section'>
  <entry_section></entry_section>
</preset_item>
```

The component class gets the preset value as follows:

```
IConfigElement presetElement = lookupElement ("preset_item[id=entry_section]");
String entrySection = presetElement.getChildValue("entry_section");
```

Troubleshooting presets

To make sure that your changes to presets have taken effect, you can force the application server to refresh by calling `refresh.jsp` (the default sync time specified in `app.xml` is 60 minutes). Calling `refresh.jsp` refreshes the static configurations and synchronizes presets from the repository before refreshing the in-memory `ConfigService` structures.

It can be helpful to enable tracing of presets, which provides information about the modifications that presets make to the `ConfigService` when you create, edit, and delete presets. To enable tracing of presets, turn on the `PRESET` and `CONFIGMODIFICATION` trace flags in the `/wdk/tracing.jsp` file.

Configuring and customizing 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 contain externalized strings in WDK:

- Properties files

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

- Accessibility image strings ([Providing image descriptions, page 453](#))

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`.

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.

The following topics describe how to internationalize applications and modify UI strings.:

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 348).

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 applet JSP tags such as dmf:filebrowse.

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 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 filename, 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"`

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.

Displaying escaped HTML strings

To avoid the risk of cross-site scripting in which a user could enter malicious JavaScript as an attribute value, attribute values are displayed exactly as they are stored, so document attributes containing markup such as `"<Work>"` will be displayed as `"<Work>"`. Document attributes with a value of `"<Work>"` will be saved and displayed as `"<Work>"`.

You can change the default 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" encodeattribute="false"/>`. Note that if you turn off the either globally or locally you put your application at risk for cross-site scripting.

This is accomplished behind the scenes by (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 [Utilities, page 449](#). For information on configuration of app.xml to detect URL parameters that are susceptible to cross-site scripting, refer to [Turning on cross-site scripting security, page 106](#).

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 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.)

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 7.3.06 for the English date March 7, 2006.
- 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 problem. Make sure the browser has fonts that can display the character of your application.
- Check for high ANSI characters, for example, , , , , which indicate the wrong code page for the application server.
- Delete generated class files for JSP pages that could contain your strings.

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

Using the internationalization debugging component

1. Add the `debug_preferences` component to the preferences component definition. Create a configuration file with the following content:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<config>
<scope>
<component modifies="
  preferences:webcomponent/config/environment/preferences/preferences_component.xml">
<insert path='contains'>
<component>debug_preferences</component>
</insert></component>
</scope></config>
```

2. Refresh the configurations in memory by restarting the application server.
3. Log in and choose **Preferences**, then choose the **Debug** tab, show in [Figure 22, page 351](#).

Figure 22. Debug preferences



4. To test for non-localized strings, set the value of `<application>.<language>.<fallback_to_english_locale>` to `false` and refresh the configuration files in memory. Then check Test NLS Strings in

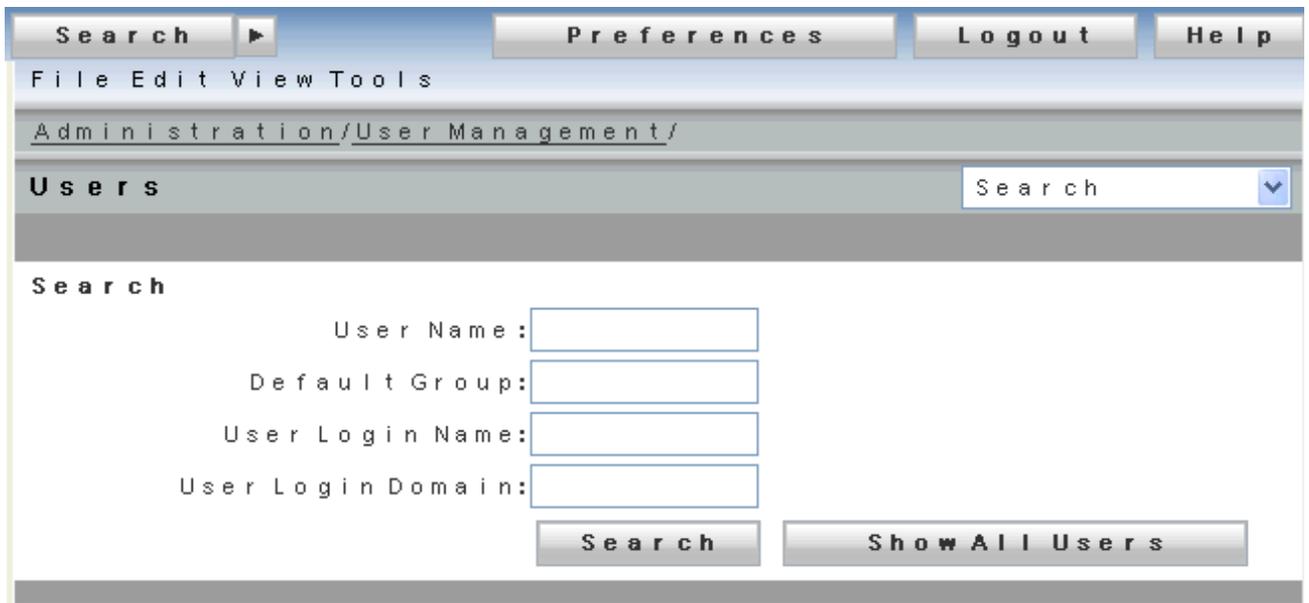
the Debug preferences. You will see `xx` surrounding each UI string for the Test NLS Strings checkbox. Strings that are provided by user input, queries, or image files are not affected. UI strings that you have not internationalized will not have the surrounding `xx`. The Username field label below was not localized, so the label is not surrounded by `xx`:



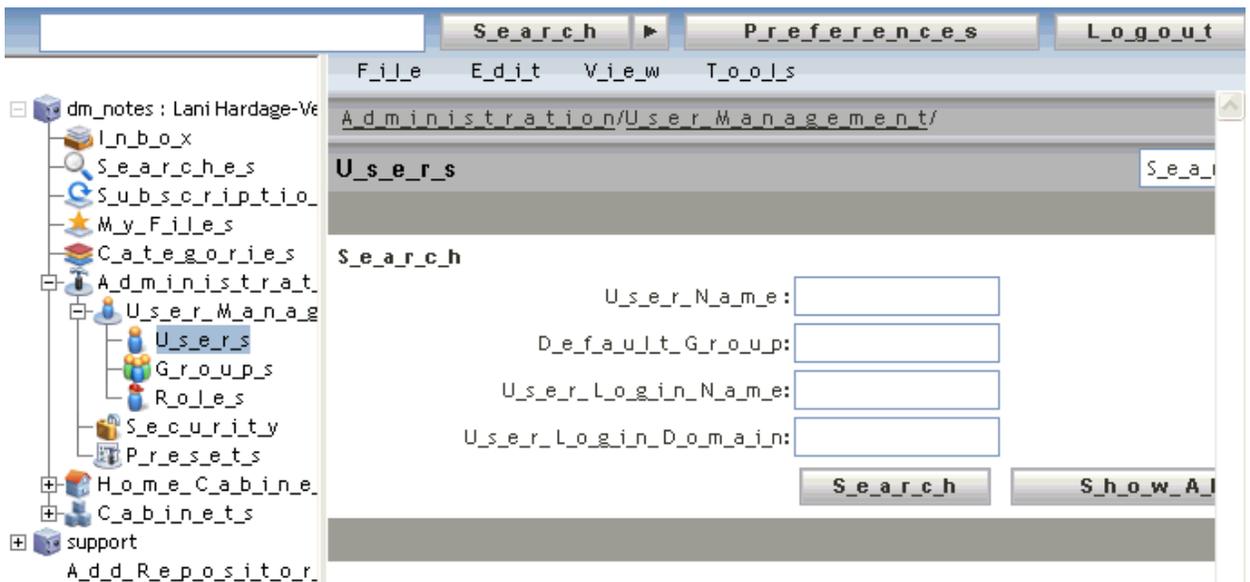
The screenshot shows a login form with a blue header and footer. The header contains the text `xxLoginxx`. The main content area has the following elements:

- Header text: `xxWebtopxx xx6.0xx`
- Username field: `USERNAME:` followed by a text input containing `hardal`.
- Password field: `xxPasswordxx:` followed by a text input.
- Repository field: `xxRepositoryxx:` followed by a dropdown menu showing `dm_notes`.
- Checkbox: `xxRemember my credentialsxx`
- Link: `[+] xxMore Optionsxx`
- Footer: A question mark icon on the left and a `xxLoginxx` button on the right.

- To test for the effect on the UI of double-byte languages, check Test Far Eastern Characters. The test displays each UI string with a space between each letter. 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. The results are similar to the following:



- To test for the effect of a small screen on the UI, check **Test Long Strings**. Test The test displays each UI string with an underscore between each letter. Strings that are provided by user input, queries, or image files are not affected. In the following example, one button is cut off on the right, and strings do not wrap in both left and right frames:



Retrieving localized strings

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 customizing strings, page 346](#).

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") %>'></span>
```

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);
```

Use the `NlsResourceClass` and `NlsResourceBundle` method `stringExists()` to determine whether a string exists.

Adding 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 7-3. 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.

`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`.

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 following topics describe how to create themes and styles to brand your application. For information on how to change the icon that is displayed for virtual document or assembly items, refer to [Configuring type icon display, page 109](#).

Creating a new theme

Each application-layer directory contains a theme directory that contains one or more 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 documentum theme in the webcomponent layer inherits all of the documentum 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 custom theme, page 357](#)).

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 application only if they contain files or other directories.

Registering a custom 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, add your custom theme to the <themes> element. Note that the <nlsbundle> element points to a custom properties file that you will create:

```
<application modifies='webtop/app.xml'>
  <insert path="themes">
    <theme>
      <name>mytheme</name>
      <base-theme>documentum</base-theme>
      <label><nlsid>MSG_BRAND_MYTHEME</nlsid></label>
    </theme>
  </insert></application>
```

Note: Because the <application> element that you are modifying does not have an ID, you must leave it out of the modifies attribute value as shown in the example.

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. By default, app.xml specifies that BrandingServiceNlsProp contains the mapping table. This element is optional and, if omitted, its value is inherited from the parent definition.
- <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). If an <nlsid> element is used, its value is resolved by a lookup in the nlsbundle properties file.

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/web/common or in the location that you have specified for the value of the <nlsbundle> element in the custom app.xml file.

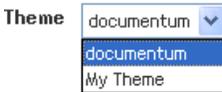
Add the following content to the custom BrandingServiceNlsProp.properties file:

```
NLS_INCLUDES=com.documentum.web.common.BrandingServiceNlsProp
MSG_BRAND_MYTHEME=My theme
```

Note: After saving your changes (for example, when adding a new theme directory), refresh the cached theme list in the application by navigating to the refresh utility page `wdk/refresh.jsp`.

The resulting drop-down theme list is similar to the following:

Figure 23. Custom theme



Note: When you add a new string to a Java properties file, you must restart the application server in order for your string to be read into the 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. The following example overrides the background color in a style defined in `webtop/theme/documentum/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/mytheme/css/custom.css`.

The list of style sheets to be applied to a JSP page is assembled at runtime. 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 `mytheme` 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/mytheme/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/mytheme/css/custom.css">
```

Style sheets in the same directory are added to the list in alphabetical order. For example, if a directory `custom/mytheme/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.

Modifying a style sheet

You can modify style sheets to change the look of applications. Style sheets override styles defined in lower application layers. For example:

- The style sheet in the `webtop/documentum/css` directory overrides the style sheet in the `webcomponent/documentum/css` directory.
- The style sheet in the `custom/documentum/css` directory overrides the style sheet in the `webtop/documentum/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' />
```

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.

Example 7-4. 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 cabinet icon is shown below. The custom icon filename is `t_dm_cabinet_16.gif`:

Figure 24. 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.

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 359](#).

Overriding object type icons — To override icon display for specific object types, open your custom `app.xml` and add the new `<iconoverrides>` element with a child `<alwaysshowicontype>` element for each repository type that should display a type-based icon. The value of this element must be the object type. For example, the following entry would render the object type `rm_dod5015ch2record` with a type-based icon rather than an assembly icon:

```
<iconoverrides>
  <alwaysshowicontype>rm_dod5015ch2record</alwaysshowicontype>
</iconoverrides>
```

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 [Providing image descriptions, page 453](#) for more information.

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

```
.webtopMessagebarBackground
{
  BACKGROUND-IMAGE: url('../images/statusbar/shadowbg.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 the correct image is used.

Configuring buttons

The button control by default renders a standard HTML button element of the form `<button>Hello</button>`. The button style is set on the button tag by a class attribute which references a CSS class. For example, see the following button control in `dqlEditor.jsp`:

```
<dmf:button nlsid='MSG_EXECUTE' onclick="onClickExecute" tooltipnlsid=.../>
```

The HTML that is rendered is the following:

```
<button type="button" name='DQLEditor_Button_0' title="Execute" class="button" onclick='setKeys(event);__x9a9onclick(this);' >Execute</button>
```

Two other types of buttons are also supported:

- **Image button**
Renders HTML `<button>` tag and `` tags. Specify the path to a button graphic file relative to the `/theme/theme_name` as the value of the `src` attribute on the button tag, for example: `<dmf:button ...src='images/mybutton.gif' />` You can specify as the value of `srcdisabled` an image to be displayed when the button is disabled. Do not use the deprecated `imagefolder` attribute.
- **Graphic button (deprecated)**
Renders HTML `<table/><tr/>`, and `<td/>` tags.

To change the text for a button

The task is to change the **Help** button text to **Resources**.

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. The Help button in the login page `login.jsp` has an attribute value of `MSG_HELP`:

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

2. Search for this string in the Java properties files to find the file. It is in the properties file for the titlebar component, but that is not the button we are looking at. It is in every component properties file, and we find it in the `Webtop LoginNlsProp.properties` file.
3. Modify the login component to use your own `nlsbundle`. For example:

```
<component modifies="login:/webtop/config/login_component.xml">  
  <replace path="nlsbundle">  
    <nlsbundle>com.mycompany.LoginNlsProp</nlsbundle></replace>  
</component>
```

4. Find the string identified by the `nlsid` key value on the control tag in the Webtop `LoginNlsProp.properties` file. The key value is `MSG_HELP=Help`.
5. Create a new properties file named `MyLoginNlsProp.properties` in the directory `WEB-INF/classes/com/mycompany`. 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,
com.documentum.webtop.session.LoginNlsProp
MSG_HELP=Resources
```

How themes are located by the branding service

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. 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 high contrast theme, the resource loader loads the first file named `/t_dm_document_16.gif` that it finds from the following search path:

```
custom/theme/high contrast/icons/type/t_dm_document_16.gif
webtop/theme/high contrast/icons/type/t_dm_document_16.gif
webcomponent/theme/high contrast/icons/type/t_dm_document_16.gif
wdk/theme/high contrast/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.

Creating asynchronous jobs

The following sections describe asynchronous support.

Asynchronous action job execution

Custom actions based on WDK 5 or higher will work in the 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 7-5. Enabling asynchronous execution of an action

This is an example of how to enable 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 or higher will work in the 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 7-6. 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 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 7-7. 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 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 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 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.

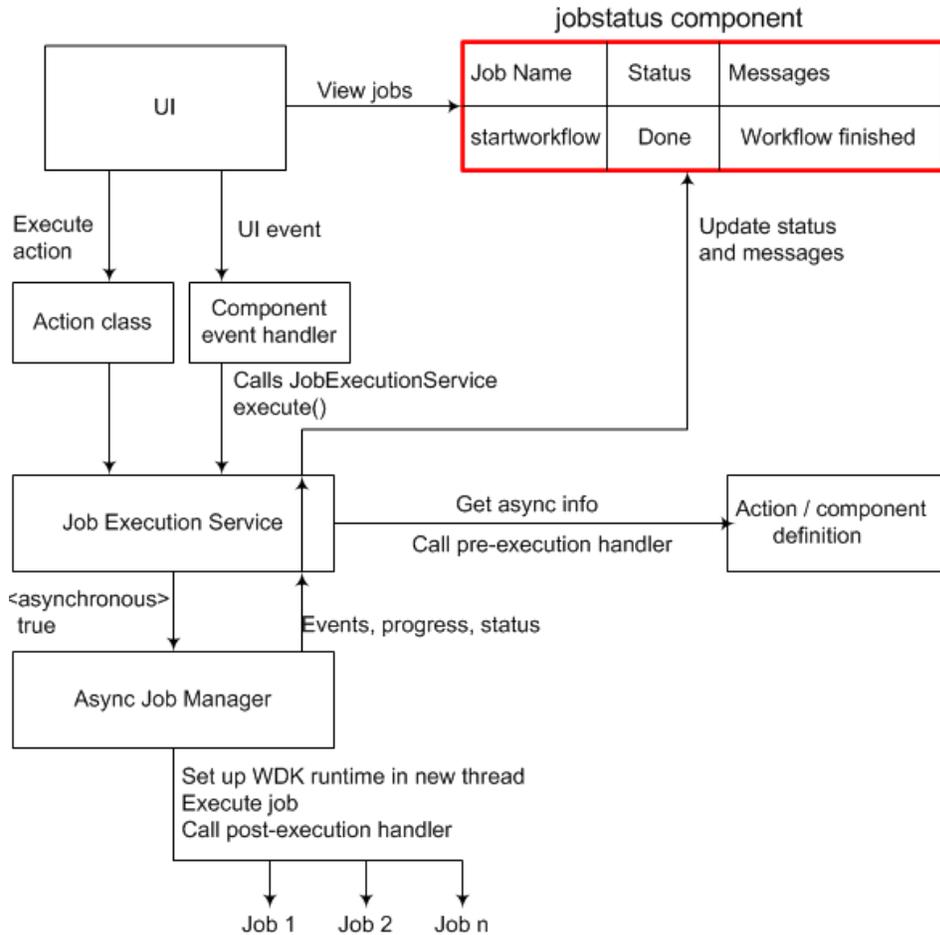
- Display the status of an asynchronous job

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

Figure 25, page 369 diagrams the process in which asynchronous actions and component jobs execute.

Figure 25. Job execution interaction diagram



Customizing authentication and 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. `ClientSessionState` differentiates the state between browser windows that are associated with a common HTTP session.

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`.

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.

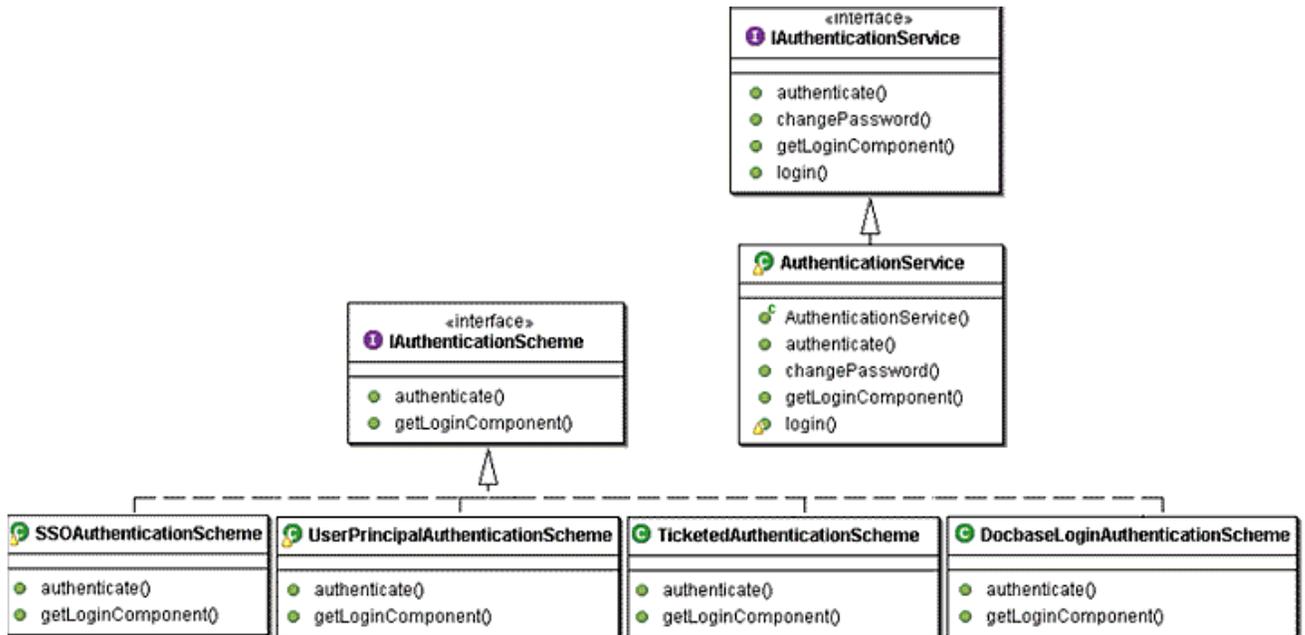
The authentication service

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 26. Authentication service interfaces



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 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 because they are a memory-cached row from a collection.

Example 7-8. 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 7-9. 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 in any WDK class

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 7-10. 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)`. If you need to launch additional sessions, save session-specific data using `ClientSessionState` instead of `SessionState`. 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 `ClientSessionState` and does not create URLs using `addBrowserIdToURL`, session state will be shared by multiple browser windows, as in previous versions of WDK. The following example stores and later retrieves a session-specific variable in `ClientSessionState`:

```
m_ticket = request.getParameter("ticket");
ClientSessionState.setAttribute(AD_HOC_TICKET, m_ticket);
...
m_ticket = (String) ClientSessionState.getAttribute(AD_HOC_TICKET);
ClientSessionState.removeAttribute(AD_HOC_TICKET);
```

Clearing DFC and WDK object caches

Caching timeouts can be set on formats and folders so new folders or locations will be seen after the cache expiration.

The cache timeout for formats is set in the file `FormatService.properties`, located in `WEB-INF/classes/com/documentum/web/formext/docbase`. The setting `cacheLifetime` specifies the cache lifetime between refreshes in seconds.

The cache timeout for folders is set in the file `FolderUtil.properties`, located in `WEB-INF/classes/com/documentum/web/formext/docbase`. The setting `cacheLifetime` specifies the cache lifetime between refreshes in seconds.

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.

Adding an application, session, or request listener

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 51, page 112](#).

Listening to DFC sessions

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 7-11. 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
}
```

Synchronizing a session

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 7-12. 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());}
}
```

Supporting multi-repository sessions

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:

- [Managing a multi-repository session, page 378](#)
- [Using replica \(mirror\), reference, and foreign objects, page 380](#)
- [Adding multi-repository support to a component, page 379](#)
- [Scoping and preconditioning actions on remote objects, page 380](#)

Managing a multi-repository session

DFC 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, do not use `Component::getDfSession()`. Instead, use `IDfSessionManager` as in the following example:

```
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);
}
```

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 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.

Using 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 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 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 determine 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 380.

Tracing sessions

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`.

Adding an authentication scheme

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 [Custom authentication](#), page 460. 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)

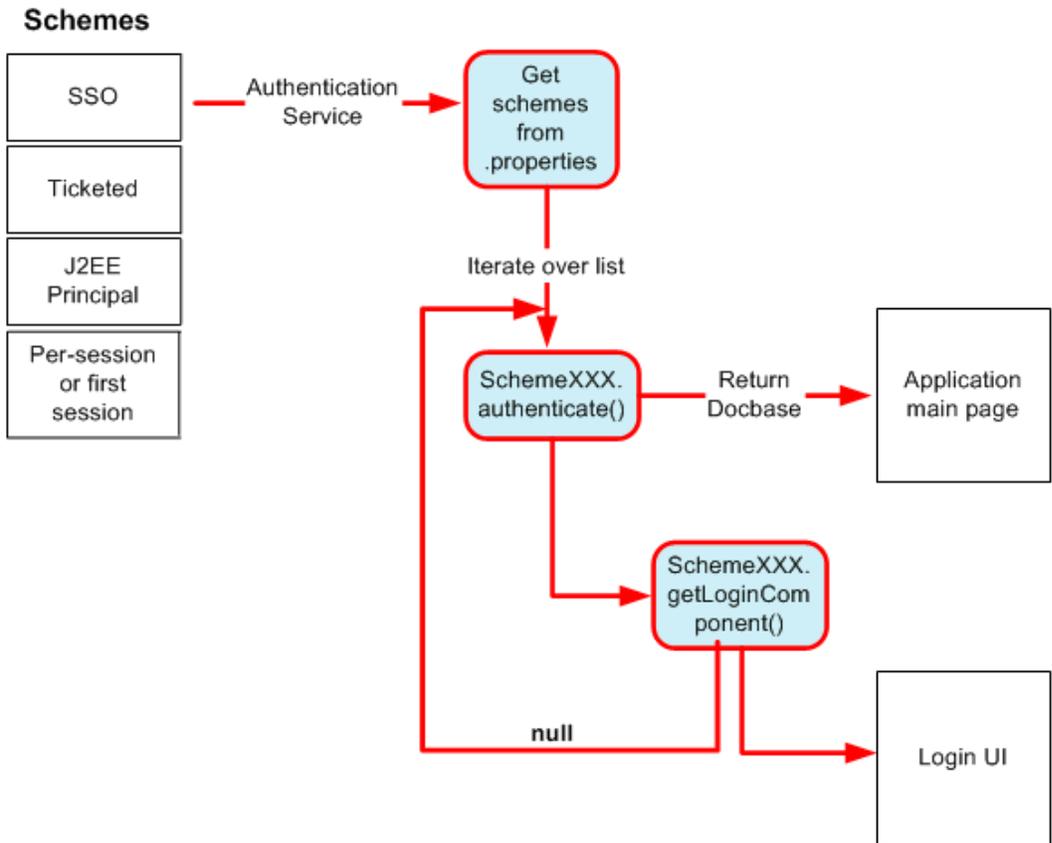


Caution: The DFC session manager does not support mixed authentication schemes. For example, if the user logs into the first repository with a user principal login and logs into the second repository with a session login dialog, the second login attempt will throw an exception.

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. Figure 27, page 382 diagrams this sequence.

Figure 27. 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`.

Supporting 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 7-13. Login from external resource

Add login properties of repository, 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 type of content. Substitute appropriate repository name, user name, password, and domain if needed.

```
repository=dm_notes
username=myname
pwd=ab8xT33
domain=
```

Create an authentication scheme that implements `IAuthenticationScheme` and reads the login properties, similar to the following class:

```
package com.mycompany.session;
import java.util.ResourceBundle;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.DfLogger;
import com.documentum.web.common.ArgumentList;
import com.documentum.web.formext.session.AuthenticationService;
import com.documentum.web.formext.session.IAuthenticationScheme;
import com.documentum.web.formext.session.IAuthenticationService;

public class AutoLoginScheme implements IAuthenticationScheme
{
    public String authenticate(HttpServletRequest request,
        HttpServletResponse response, String doctype) throws DfException
    {
```

```
    System.out.println("In AutoLoginScheme");
    ResourceBundle bundle = ResourceBundle.getBundle("
com.mycompany.session.AutoLogin");
    if (bundle != null)
    {
        String strRepository = bundle.getString("repository");
        System.out.println("repository in properties file: " + strRepository);
        String strUsername = bundle.getString("username");
        System.out.println("username in properties file: " + strUsername);
        String strPwd = bundle.getString("pwd");
        String strDomain = bundle.getString("domain");
        System.out.println("domain in properties file: " + strDomain);

        IAuthenticationService service = AuthenticationService.getService();
        HttpSession sess = request.getSession();
        if (sess != null)
        {
            System.out.println("obtained session");
            service.login(sess, strRepository, strUsername, strPwd, strDomain);
            System.out.println("Returned from service.login");
            return strRepository;
        }
        else
        {
            DfLogger.debug(this, "HTTP session was null", null, null);
            return null;
        }
    }
    else
    {
        DfLogger.debug(this, "ResourceBundle was null", null, null);
        return null;
    }
}
public String getLoginComponent(HttpServletRequest request,
    HttpServletResponse response, String doctype, ArgumentList outArgs)
{
    return null;
}
}
```

After you have compiled your class, two more steps are required to implement your custom login scheme:

1. Edit the file `AuthenticationSchemes.properties`, which is located in the directory `WEB-INF/classes/com/documentum/web/formext/session`. Place your custom login class first in the list, for example:

```
scheme_class.1=com.mycompany.session.AutoLoginScheme
scheme_class.2=com.documentum.web.formext.session.TicketedAuthenticationScheme
scheme_class.3=com.documentum.web.formext.session.RSASSOAuthenticationScheme
...
```

2. Restart the application server.

Configuring and Customizing Search

The following topics describe search configuration and customization for WDK-based applications.

Understanding search in WDK

Following is a brief general description of the WDK customization model. Information on individual search controls and components is contained in the comprehensive reference guide, *Web Development Kit and Webtop Reference Guide*.

WDK and Webtop search components can search multiple repositories and external sources. The search components are versioned, so if a request is made for a search component, the new component is returned by default. If you customized a previous version of a Webtop search component and extended it, and the version is still supported, your customization will be used in place of the new search components.

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 — All attributes are indexed, so a query for attribute criteria is run against the full-text index by default. 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. Verity operators such as "not" or "between" are not supported. .

Tip: You can specify in the data dictionary the default operator to be used for an attribute, such as "is" or "is not".

Saving searches — Searches are saved as smartlist objects. Saved searches save the display configuration as well as the query, and the user has the option of saving query results with the query. Users can revise a saved search using the advanced search component.

Smartlists created with Documentum Desktop can be executed or edited in the advanced search UI but will no longer be usable in Desktop. Smartlists that are created in WDK applications cannot be used or edited in Desktop.

The `saveSearch` component displays checkboxes that allow the user to save search results with a search and to make the saved search public. These two features can be removed by setting the value of the configuration element `<enablesavingsearchresults>` to `false`. The following example in a modification file removes these two checkboxes:

```
<component modifies="
  saveSearch:webcomponent/config/library/saveSearch/saveSearchHex/saveSearch_component.xml">
  <replace path="enablesavingsearchresults">
    <enablesavingsearchresults>false</enablesavingsearchresults>
  </replace></component>...
```

The default for whether to save results with a search is set by the configuration element `<includeresults>`.

Full-text search — Simple and advanced searches query the full-text index by default. You can run a full-text query in advanced search using the **Contains** field. The **Contains** field or the simple search text box can contain a string within quotation marks to search for the exact string, for example, "this string". The box also supports the operators AND and OR operators. The following rules apply:

- Either operator may be appended with NOT.
- The operators are not case-sensitive.
- Punctuation, accents, and other special characters are ignored (replaced with a space).
- The AND operator has priority over the OR operator. For example, if you type `knowledge AND management OR discovery`, the results will contain both `knowledge` and `management` or that contain `discovery`.
- Parentheses override the priority of operators. For example, if you type `knowledge AND (management OR discovery)`, the results must contain `knowledge` and must also contain either `management` or `discovery`. The NOT operator cannot be used to qualify an expression within parentheses, for example, `NOT (a and b)`, but it can be used within parentheses, for example `a OR (b and NOT c)`.
- If no operators are used between words, multiple words are treated with the ACCRUE operator, which is an OR operator with a result ranking that gives a higher score to results that contain all words. The NOT operator is processed differently.

The wild card operator `*` allows 0 or more characters added to the string in the text box, for indexed repositories only, and the operator `?` allows a single character.

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. For information on supporting conditional value assistance in JSP pages, refer to [Providing conditional value assistance, page 392](#).

Limitations:

- Not all values in value assistance may be available across repositories in a logical OR operation. (This is not a limitation for the AND operation.)
- Locale-based assistance values are not available in a search of repositories on multiple locales unless the assistance values are present in each locale's data dictionary.

Presets for search on object type — In the Webtop presets editor, you can create a preset that limits the searchable object types. This preset will override the <includetypes> setting in the advanced search component definition.

Webtop Extended Search — Webtop extended search can be installed on a Content Server to provide search results clustering, search templates, and search monitoring. The clustering DocApp must be installed in a global registry repository of version 6. This DocApp also enables search monitoring. The search templates DocApp must be installed in each repository (version 5.3.x or 6) in which you wish to store search templates. Instructions for installing the Webtop Extended Search DocApps are in the Web Development Kit and Webtop Deployment Guide.

Configuring search controls

Refer to Web Development Kit and Webtop Reference Guide for details on each control's configuration.

You can globally configure all instances of certain advanced search controls by modifying the control configuration definitions on `wdk/config/advsearchex.xml`. The following controls can be configured:

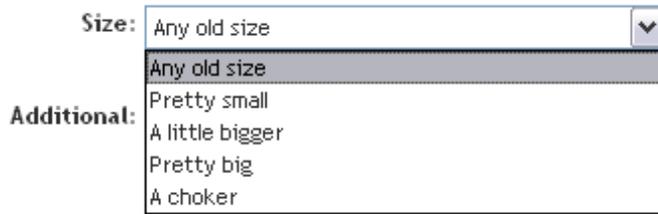
- match case attribute on search attribute controls (does not apply to searches of the index)
- searchsizeattribute control
- searchdateattributecontrol
- search clusters

The following example changes the size range dropdown selections. It modifies `advsearchex.xml` in a modification file located in `custom/config` with the following content:

```
<config version='1.0'>
  <scope type='dm_sysobject'>
    <searchsizeattributerange modifies="searchsizeattributerange:
      wdk/config/advsearchex.xml">
      <insert>
        <option>
          <label>Any old size</label>
          <operator>LT</operator>
          <value>-1</value>
          <unit>KB</unit>
        </option>
      </insert>
    </searchsizeattributerange>
  </scope>
</config>
```

The resulting UI shows the new values for size attribute range:

Figure 28. Search size custom drop-down list



Case sensitivity — Search on full-text strings or attributes against a 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. If you turn off case sensitivity, create functional indexes on the attributes that will be queried.

If you use a `ENABLE(NOFTDQL)` hint in a DQL hints file to send attribute queries against the database rather than the full-text index, your query will be case-sensitive if you set the value of `<defaultmatchcase>` to `true`. For better performance, set case sensitivity to `true`, or set it to `false` and create a functional index on the queried attribute columns.

Configuring basic search

Basic search searches all sysobjects in the current repository for the user-supplied string in the full-text index of content and attributes. 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.

Users can search attributes only and not full text in the following ways:

- Add a checkbox for **Include recently modified properties** on the advanced search page. Attributes will be queried against the metadata and not the index. To do this, uncomment the following lines your custom advanced search JSP page (a copy of the webcomponent advanced search JSP page):

```
<!--
<tr class="leftAlignment" valign=top>
<td class="leftAlignment" valign=top nowrap>
<dmfxs:searchscopecheckbox name='<%=AdvSearchEx.DATABASE_SEARCH_SCOPECHECKBOX_CONTROL%>'
  scopename='<%=RepositorySearch.DATABASE_SEARCH_PROPERTY%>' checkedvalue='true'
  uncheckedvalue='false' nlsid='MSG_DATABASE_SEARCH' tooltipnlsid="MSG_DATABASE_SEARCH_TIP"/>
</td>
</tr>-->
```

- Use the dql query type for a custom search component and pass the query string in the query parameter.
- Turn off FTDQL (queries against index) using a DQL hints file. You can disable index queries for attributes without affecting the full-text string portion of a query. For more information, refer to .

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:

- The 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 "=" and "<>" search operations, if defined in the data dictionary

The WDK search UI contains search controls. To make changes to control attribute values, you must extend a search component and modify your custom search JSP page.

Example 8-1. Setting the search type drop-down list

The available search types list is configured by the `includetypes` element in the `advsearch` component definition. The `includetypes` list is comma-delimited. The `descend` attribute specifies whether subtypes or included or not. Create your modification definition in `custom/config`. The following example will display `dm_folder` and all of its subtypes including custom types that subtype `dm_folder`:

```
<component modifies="advsearch:webtop/config/
advsearchex_component.xml">
  <replace path="includetypes">
    <includetypes descend="true">dm_folder</includetypes>...
  </replace></component>
```

Figure 29, page 392 shows the type selection list set by `includetypes` with `descend` set to `true`:

Figure 29. Limiting the selectable types and subtypes

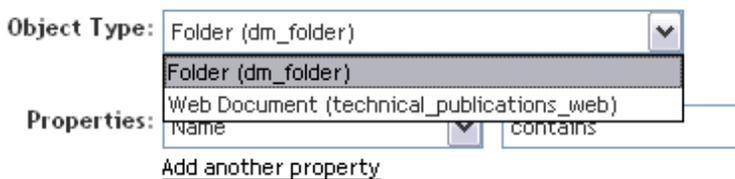


The following example will display only two selections, because the descend parameter is set to false:

```
<includetypes descend="false">
  dm_folder, my_type
</includetypes>
```

Figure 30, page 392 shows the type selection list set by includetypes with descend set to false:

Figure 30. Limiting the selectable types without subtypes



Example 8-2. Providing conditional value assistance

Use individual searchattribute control tags to provide conditional value assistance. The searchattributegroup tag provides only simple attribute assistance.

The lists of conditional values are set in Documentum Application Builder. Query value assistance can use a reference (`$value(attribute)`), for example:

```
SELECT "MyDocbase"."MyTable"."MyColumn1" FROM "MyDocbase"."MyTable"
WHERE "MyDocbase"."MyTable"."MyColumn2" = '$value(MyAttribute)'
```

The following example lists four attributes, three of which have conditional value assistance lists that were set up in Documentum Application Builder. The drop-down list for Make determines the list available for Model. The drop-down lists Fuel and Year both depend on Model.

Figure 31. Conditional value assistance UI

Location: ORA53SP1ECI7

ECI_car_type (eci_car_type) ▾

Make: = ▾	Ferrari ▾
Model: = ▾	Testarossa ▾
Year: = ▾	Testarossa
Fuel: = ▾	F40
	Unleaded ▾

This UI was generated from the following set of controls in the JSP page:

```
<tr>
  <td>Make:</td>
  <td><dmfxs:searchattribute name='make' attribute="make"/></td>
</tr>
<tr>
  <td>Model:</td>
  <td><dmfxs:searchattribute name='model' attribute="model"/></td>
</tr>
<tr>
  <td>Year:</td>
  <td><dmfxs:searchattribute name='year' attribute="year"/></td>
</tr>
<tr>
  <td>Fuel:</td>
  <td><dmfxs:searchattribute name='fuel' attribute="fuel"/></td>
</tr>
```

Example 8-3. Enabling the wild-card CONTAINS operator for string property searches

The Content Server version 6 treats the CONTAINS operator as CONTAINS WORD to avoid wild-card query timeouts against the full-text index. If the user searches on a string value in a property on the advanced search page, for example, "wdk", this string is treated as a string " wdk " (note the spaces), so that documents named "wdkindex" or "search_wdk" would not be returned, but documents named "wdk treats" or "About WDK" would be returned. Most queries that users make are for whole words, not parts of words.

You can revert to the wild-card CONTAINS operator behavior by enabling a checkbox in the advanced search JSP page. This checkbox will query the database rather than the full-text index for property strings. Recently modified properties that have not yet been indexed will also be returned.

Note: The properties query against the database will be case-sensitive.

To enable the checkbox, remove the JSP comment tags around the following tag in the advancedsearchex.jsp page:

```
<!--  
<tr class="leftAlignment" valign=top>  
<td class="leftAlignment" valign=top nowrap>  
<dmfxs:searchscopecheckbox name='<%=AdvSearchEx.DATABASE_SEARCH_SCOPECHECKBOX_CONTROL%>'  
  scopename='<%=RepositorySearch.DATABASE_SEARCH_PROPERTY%>' checkedvalue='true'  
  uncheckedvalue='false' nlsid='MSG_DATABASE_SEARCH' tooltipnlsid="MSG_DATABASE_SEARCH_TIP"/>  
</td>  
</tr>-->
```

Configuring search results and tuning performance

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, globally and per source, is configured in dfc.properties. The maximum number of search results is specified as the value of dfc.search.maxresults (was maxresults_per_source in 5.3.x). The maximum number of results per source is specified as the value of dfc.search.max_results_per_source. For example, if you have specified a maximum of 1000 results and a maximum per source of 500, results will be accumulated from each source until the source maximum of 500 is reached or until the global maximum of 1000 is reached.

Note: These settings can affect performance. Setting the value too high can overload the index server, and setting it too low can frustrate users. You must evaluate the best settings for your environment.

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.

Folder path display is disabled for faster performance — The display of the results' folder path is turned off in order to enhance query performance. The value of <displayresultspath> in webcomponent/config/library/search/searchex/search60_component.xml is set to false.

Turning off summary calculation for faster performance — The summary column is calculated, which can add to query overhead. Turn off the summary column by extending the Webtop search component `searchex_component.xml` in `webtop/config`. Copy the `<columns_XXX>` elements (`<columns_drilldown>`, `<columns_list>`, and `<columns_saved_search>`) from the parent configuration file `search60_component.xml` in `webcomponent/config/library/search/searchex`. In each of the `<columns>` elements, set the value of `<column>.<attribute>.<visible>` for the summary attribute to `false`. Set the value of `<columns_XXX>.<loadinvisibleattribute>` to `false` to ensure that the column is not calculated.

Configuring the display of attributes in search results — Default search result columns are configured as `<column>` elements in the basic search configuration file `search60_component.xml` in `webcomponent/config/library/search/searchex`. Only attributes marked as searchable in the data dictionary 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 default 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 and Webtop Reference*

In the following simple configuration, the definition extends the WDK search component definition and adds some custom attribute columns:

```
<config version='1.0'>
<scope type='technical_publications_web'>
  <component modifies=""
    search:webcomponent/config/library/search/searchex/search60_component.xml">
    <insert path='columns_list'>
      <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>
    </insert>
  </component>
</scope>
</config>
```

Making custom attributes available in search results

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. For example, calculated columns such as score or summary will not display any values unless they are selected before the query is run.

You must modify the definition for the `display_preferences` component to make columns of your custom type available to users for display.

To make a custom type available in preferences

1. Modify the `display_preferences` component in your custom/config directory: :

```
<component modifies="
  display_preferences:webtop/config/display_preferences_ex_component.xml">
```

2. Add your custom type to the `<display_docbase_types>` element. For example:

```
<insert path='preferences.display_docbase_types'>
  <docbase_type>
    <value>my_custom_type</value>
    <label>My type</label>
  </docbase_type>
</insert>
```

3. Save this file and refresh the configuration files on the application server by navigating to `wdk/refresh.jsp`.

To make a calculated attribute available in search results

1. Extend the `Search60` class in the package `com.documentum.webtop.webcomponent.search`.
2. Override `initAttributes()` and add your computed attribute. The following example adds "myComputed" attribute:

```
protected void initAttributes()
{
  List<String> mandatoryAttrs = getAttributesManager().getMandatory();
  mandatoryAttrs.add("myComputed");
  getAttributesManager().setMandatory(mandatoryAttrs);
  super.initAttributes();
}
```

3. Extend the search component definition to use your custom class, and scope it to your custom type. Set the class to use the custom class.

Customizing search in Webtop applications

The following methods in the basic search component class `Search60` provide customization points:

- `initSearch(arg)`: Override to modify queries before execution
- `initControls(arg)`: Override to update custom controls

- `initAttributes()`: Override to perform specific treatment for columns. Use `getAttributesManager()` to manipulate columns and query attributes
- `initResultsSet()`: Override to manipulate the results that are fed to the datagrid
- `initSearchExecution()`: Start the actual query execution

The following topics describe some customizations to Webtop search components.

Modifying the search JSP pages

Changes to JSP pages are considered to be customizations. The following examples extend Webtop search component definitions and specific a custom JSP page in which to make customizations.

Example 8-4. Setting the default search type

To set the default search type, supply your preferred type in the JavaScript function that calls the advanced search container. In Webtop, this is the file `titlebar.jsp`, so extend the `titlebar` component and provide the following `postComponentNestEvent` calls in the `onClickAdvancedSearch` JavaScript function. Substitute your custom type (in quotation marks) for `custom_type`:

```
postComponentNestEvent(null, "advsearchcontainer", "content", "component", "
advsearch", "type", custom_type, "usepreviousinput", "false", "query", strValue);
...
postComponentNestEvent(null, "advsearchcontainer", "content", "component", "
advsearch",
"type", custom_type, "usepreviousinput", "true");
```

Example 8-5. Displaying specific attributes for search

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:

```
...
<dmfxs:searchobjecttypedropdownlist name='objecttypectrl'.../>
</td>
</tr>

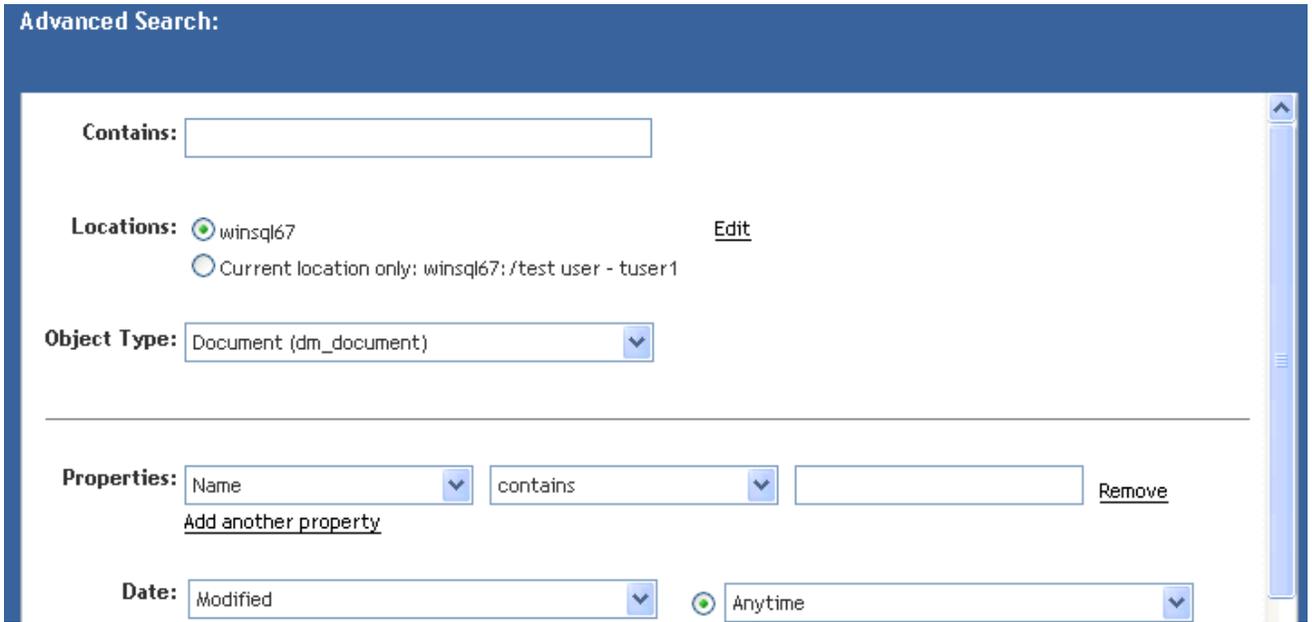
<tr><td colspan='2' class='spacer' height='10'>&nbsp;</td></tr>
<tr>
  <td align=right valign=top nowrap><dmf:label label='Name' cssclass="
  fieldlabel"/></td>
  <td align=left valign=top nowrap>
    <dmfxs:searchattribute name='searchname' attribute='object_name'
    andorvisible="false" removable="false">
    </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'
```

```

        andorvisible="false" removable="false">
        </dmfxs:searchattribute>
    </td>
</tr>...
```

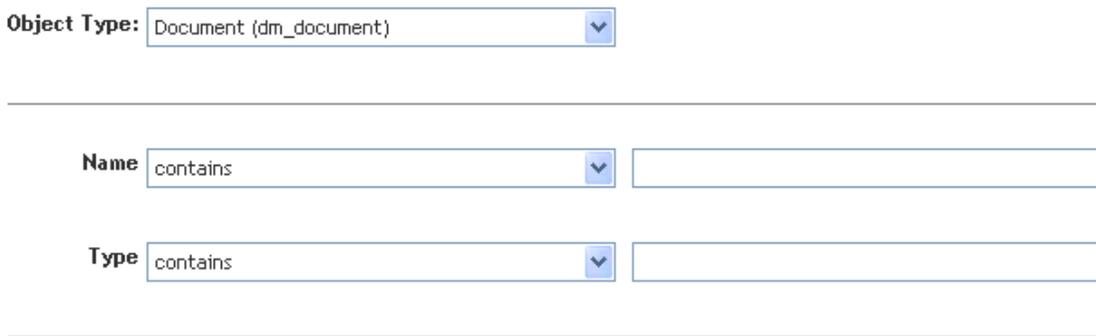
Note: Set the andorvisible and removable attributes to false on the searchattribute control. Before this customization, the user must select properties from a dropdown:

Figure 32. Attribute selection drop-down



After customization, the UI shows the individual attributes "Name" and "Type" as search criteria:

Figure 33. Specific attributes as search criteria



Example 8-6. Displaying custom attributes as search criteria

To display specific custom attributes as individual 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 34. Custom attributes as search criteria

The image shows a search criteria form with three rows. The first row is labeled 'Name' and has a dropdown menu with 'contains' selected and an empty text input field to its right. The second row is labeled 'Edition' and also has a dropdown menu with 'contains' selected and an empty text input field to its right. The third row is labeled 'Date:' and has a dropdown menu with 'Modified' selected and no text input field to its right.

Example 8-7. Enabling search on string fragments

The Content Server 6 treats the CONTAINS operator as CONTAINS (WORD) to avoid wild-card query timeouts against the full-text index. Following are two examples of the old CONTAINS and the new CONTAINS (WORD) queries:

- If the user searches on a property string in the advanced search page, such as "wdk" in the object name, this string is treated as a word with spaces before and after: " wdk ". The documents named "wdkindex" or "search_wdk" would not be returned, but the documents named "wdk treats" or "About WDK" would be returned.
- If the user searches on a property string such as "ran" in the old CONTAINS mode, "Restaurants.txt", "Insurance.pdf", "Ran by Kurosawa.doc" or "On the Run.pdf" would be returned. (Note that the last example requires lemmatization configured on the index server.) In the new CONTAINS (WORD) mode, only the last two examples would be returned.

Most queries that users issue are for whole words, not parts of words. With the new CONTAINS (WORD) operator, the query will return the same results for string properties entered into the full-text search box or a property box .

With the old CONTAINS string fragment search, lemmatization contributed to timeout errors by multiplying the number of results, but with CONTAINS (WORD), lemmatization becomes helpful.

There are three ways to revert to the wild-card CONTAINS search for a string fragment:

- Globally
 - Add 'fds_contain_fragment' to the parameter list in dm_ftengine_config object and set it to 'true'.
- For certain query conditions
 - Using the DQL hint ft_contain_fragment to a DQL hints file.

- In the Webtop UI

Uncomment the checkbox in the advanced search JSP page `advsearchex.jsp` located in `webcomponent/library/advancedsearch`:

```
<!--
<tr class="leftAlignment" valign=top>
  <td class="leftAlignment" valign=top nowrap>
    <dmf:searchscopecheckbox
      name='<%=AdvSearchEx.DATABASE_SEARCH_SCOPECHECKBOX_CONTROL %>'
      scopename='<%=RepositorySearch.DATABASE_SEARCH_PROPERTY%>'
      checkedvalue='true' uncheckedvalue='false'
      nlsid='MSG_DATABASE_SEARCH' tooltipnlsid="MSG_DATABASE_SEARCH_TIP" />
    </td>
  </tr>
-->
```

Example 8-8. Removing full-text search from the UI

This example modifies the advanced search component definition to display a custom page with specific attributes for search. To remove the full-text search or other portions of the UI, modify the search component definition that contains the JSP page specification. The top layer search component is in `webtop/config`, but that definition does not contain a JSP `<pages>` element. It extends the webcomponent advanced search definition, which does contain a `<pages>` element, so you must modify that definition.

```
<component modifies="advsearch:webcomponent/config/library/search/searchex/
  advsearch_component.xml">
```

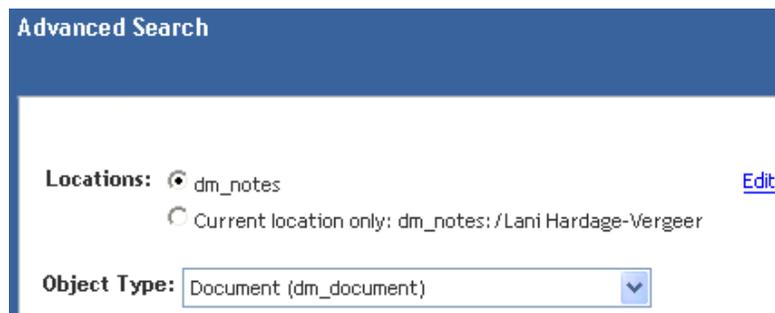
Modify the `<pages>` element to specify a custom JSP directory, similar to the following:

```
<replace path='pages.start'>
  <start>/custom/advancedsearch/advsearchex.jsp</start>
</replace>
```

Copy the Webtop advanced search JSP page `advsearchex.jsp` from `webcomponent/library/advancedsearch` to `custom/advancedsearch` and remove the table row that contains the fulltext search control:

```
<tr>
  <td scope='row' class="rightAlignment" valign=top nowrap>
    <dmf:label nlsid='MSG_CONTAINING_WORDS' cssclass="fieldlabel"/></td>
  <td class="leftAlignment" valign=top nowrap>
    <dmf:searchfulltext name='fulltextctrl' size="49"/></td></tr>
```

To remove the full-text search box for a certain object type or role, scope the definition to the desired condition. In the following example, a custom JSP page displays no full-text search box when the custom type is selected. This requires an advanced search component definition that is scoped to the custom type and that references the custom JSP page. [Figure 35, page 401](#) displays the results when the full-text search box is removed from the advanced search page:

Figure 35. Full-text search box removed from UI

Modifying the search component query

You can access a query before it is submitted and modify it in various ways. The query is accessible by overriding the `initSearch()` method of the `Search60` class. Your custom class should extend the `Webtop` version of either the `Search60` or `AdvSearchEx` component class.

Note: Custom modifications to the query will be visible to the user when they edit the query after running it.

Example 8-9. Adding a WHERE clause to simple search

To add a `WHERE` clause to the query in simple search, extend `Search60` in the package `com.documentum.webtop.webcomponent.search`. You can add criteria other than keywords to this method. If you override `buildQuery`, you may break smartlist usage. The following example adds an `AND` clause to the query that requires a specific string in the name of the object, in addition to any criteria entered into the simple search text box by the user.

First, create your extended search component definition in `custom/config` as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<config version='1.0'>
  <scope>
    <component modifies="search:webtop/config/search60_component.xml">
      <replace path='class'>
        <class>com.mycompany.SearchEx</class>
      </replace>
    </component>
  </scope>
</config>
```

Next, create your custom class that extends `Search60` and overrides `initSearch()`:

```
package com.mycompany;
import com.documentum.fc.client.search.IDfExpressionSet;
import com.documentum.fc.client.search.IDfQueryBuilder;
import com.documentum.fc.client.search.IDfSimpleAttrExpression;
import com.documentum.fc.common.IDfValue;
import com.documentum.web.common.ArgumentList;
```

```
import com.documentum.webcomponent.library.search.SearchInfo;
public class SearchEx extends com.documentum.webtop.webcomponent.search.Search60
{
    protected void initSearch (ArgumentList args)
    {
        super.initSearch(args);
        String queryType = args.get(ARG_QUERY_TYPE);
        if ((queryType == null) || (queryType.length() == 0) ||
            (queryType.equals("string")))
        {
            SearchInfo info = getSearchInfo();
            IDfQueryBuilder qb = info.getQueryBuilder();
            IDfExpressionSet rootSet = qb.getRootExpressionSet();
            IDfExpressionSet setAnd = rootSet.addExpressionSet
                (IDfExpressionSet.LOGICAL_OP_AND);
            setAnd.addSimpleAttrExpression("r_modifier", IDfValue.DF_STRING,
                IDfSimpleAttrExpression.SEARCH_OP_CONTAINS, true, false, "tuser");
        }
    }
}
```

This example adds an AND criterion in which the modifier attribute must contain the user name "tuser". Before the customization, a search on the string "Target" in the simple search box returns 3 results as shown [Figure 36, page 402](#):

Figure 36. Simple search without added clause

Search Results
 3 Results for "Target" in dalinuxora - Executed on 6/12/07 2:29 PM

Name	Modifier	Ranking	Modified	Source
TargetSetup.Result	dmadmin	73%	6/6/2007 5:52 AM	dalinuxora
Target_logo2.jpg	tuser1	70%	6/12/2007 1:41 PM	dalinuxora
Replica Target	dmadmin	72%	6/11/2007 1:42 AM	dalinuxora

After customization, only a single result in which the object name contains "Target" and the user name contains "tuser" returned. (User name is displayed in the second column, as "Modifier.")

Figure 37. Simple search with added clause

Search Results
 1 Results for "Target" in dalinuxora - Executed on 6/12/07 2:27 PM

Name	Modifier	Ranking	Modified	Source
Target_logo2.jpg	tuser1	73%	6/12/2007 1:41 PM	dalinuxora

With `IDfExpressionSet`, you can add the following operators: `LOGICAL_OP_AND`, `LOGICAL_OP_DEFAULT` (default operator in data dictionary), and `LOGICAL_OP_OR`. The following expressions, also called predicates, are available for `IDfSimpleAttrExpression` (names are self-explanatory):

```
SEARCH_OP_BEGINS_WITH
SEARCH_OP_CONTAINS
SEARCH_OP_DOES_NOT_CONTAIN
SEARCH_OP_ENDS_WITH
SEARCH_OP_EQUAL
SEARCH_OP_GREATER_EQUAL
SEARCH_OP_GREATER_THAN
SEARCH_OP_IS_NOT_NULL
SEARCH_OP_IS_NULL
SEARCH_OP_LESS_EQUAL
SEARCH_OP_LESS_THAN
SEARCH_OP_NOT_EQUAL
```

The following expression is available for `IDfValueRangeAttrExpression`:

```
SEARCH_OP_BETWEEN
```

The following expressions can be used with `IDfValueListAttrExpression`:

```
SEARCH_OP_IN
SEARCH_OP_NOT_IN
```

Example 8-10. Adding a WHERE clause to advanced search

In advanced search, you override `buildQuery` to access the user's query. The search class is as follows:

```
package com.mycompany;
import com.documentum.fc.common.IDfValue;
import com.documentum.fc.client.search.IDfSimpleAttrExpression;
import com.documentum.fc.client.search.IDfExpressionSet;
import com.documentum.fc.client.search.IDfQueryBuilder;

public class AdvSearchEx extends
    com.documentum.webtop.webcomponent.advsearch.AdvSearchEx
{
    protected IDfQueryBuilder buildQuery() throws Exception
    {
        IDfQueryBuilder qb = super.buildQuery();
        IDfExpressionSet rootSet = qb.getRootExpressionSet();
        IDfExpressionSet setAnd = rootSet.addExpressionSet
            (IDfExpressionSet.LOGICAL_OP_AND);
        setAnd.addSimpleAttrExpression("object_name", IDfValue.DF_STRING,
            IDfSimpleAttrExpression.SEARCH_OP_CONTAINS, true, false, "xpath");
        return qb;
    }
}
```

Example 8-11. Changing the query source

You can change the location, including the source and folder path in the repository with query builder APIs. The following example adds a source repository to IDfQueryBuilder instance and sets a path within the repository for the query. The examples for basic and advanced search show you how to get the query builder instance (variable qb in this example):

```
qb.clearSelectedSources();
qb.addSelectedSource("dm_notes");
// set source, path, descend flag
qb.addLocationScope("dm_notes", "/Temp", false);
```

The resulting query is similar to the following:

```
SELECT r_object_id,text,object_name,FROM dm_document
SEARCH DOCUMENT CONTAINS testing WHERE (object_name
LIKE %testing% ESCAPE \) AND FOLDER(/Temp) AND (a_is_hidden = FALSE)
```

Hiding the customization from query editing

If you have intercepted and modified a query after form submit, the hidden query processing will be displayed when the user tries to modify the query. To hide the custom modification, add the `usepreviousinput` parameter in the call to the advanced search component. Modify the titlebar component definition to use your own titlebar.jsp page as follows:

```
<component modifies="titlebar:webtop/config/titlebar_component.xml">
  <replace path="pages.start">
    <start>/custom/titlebar/titlebar.jsp</start>
  </replace></component>
```

In your custom titlebar JSP page, change the call to the advanced search component to set `usepreviousinput` to false:

```
postComponentNestEvent(null, "advsearchcontainer","content","advsearch",
  "type", "dm_sysobject", "usepreviousinput", "false")'
```

Programmatic search value assistance

Data dictionary value assistance is available in advanced search. 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 drop-down list with all existing values for the selected object type
- `ObjectTypeVATag`
Populates the search object type drop-down 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);
}
```


Configuring and Customizing Content Transfer

The following topics describe some common content transfer customization tasks:

- [Content transfer modes compared, page 407](#)
- [HTTP content transfer, page 410](#)
- [Streaming content to the browser, page 412](#)
- [UCF overview, page 412](#)
- [Initializing content transfer controls, page 412](#)
- [Configuring UCF components, page 413](#)
- [Customizing a UCF component, page 414](#)
- [Troubleshooting UCF, page 415](#)
- [Logging UCF, page 416](#)
- [Getting return values from content transfer, page 418](#)
- [Content transfer service classes, page 419](#)
- [Displaying content transfer progress, page 420](#)
- [Logging and tracing content transfer component operations, page 420](#)

Content transfer configuration tasks are generally done as administration of an application. For information on configuration of content transfer, refer to [Configuring content transfer options, page 47](#).

Content transfer modes compared

[Table 84, page 408](#) compares feature support for the two modes of content transfer.

Note: UCF content transfer is not used for import when the user has selected accessibility mode. HTTP content transfer is used in accessibility mode.

Table 84. Feature support in content transfer modes

Feature	UCF	HTTP
Download to client	Small applet with client-side UCF deployment	No client-side deployment
Drag and drop	Supported on IE browser	Not supported
Viewing or editing application	Configurable	Controlled by browser
ACS support	Supported	Limited support for export or edit; not supported for view with relative links
Progress display	Supported	Supported only by certain browsers
Preferences	Supported	Not supported
Restart interrupted operation	Supported	Not supported
Checkout	Supported	Limited support. User must save and select checkout location.
Edit	Supported	Limited support. User must save and select checkout location.
Checkin	Supported	Limited support. User must navigate to saved document.
View	Supported; does not transfer content if file is up to date on client	Supported; always transfers content
Export	Supported	Supported
Import	Supported	Limited support. Single file selection at a time, no folder import.
Client xfer tracing	Supported	Not supported
Server xfer tracing	Supported	Supported
File compression	Supported, with configurable exceptions	Turn on HTTP compression in web.xml
XML application	Supported	Import single file against Default XML Application
Virtual document	Supported	Root only

Table 85, page 409 describes the client configuration settings for UCF and HTTP modes.

Table 85. Client configuration settings in content transfer modes

Setting	<option name=> (client.config.xml)	HTTP
Temp working directory for upload/download	temp.working.dir	None
Checked out files	checkout.dir	User must save file, then check in from file
Exported files	export.dir or user choice	Browser-specific UI to select a download location
Viewed files	viewed.dir	Browser temporary internet files location, for example, \$java{user.home}\Local Settings\Temporary Internet Files
User location (base for other locations)	user.dir (defined in config file)	None
Registry file location	registry.file	None
Registry mode	registry.mode	None
Log file location	logs.dir	None
Tracing/debug	tracing.enabled	None
File polling	file.poll.interval	None
Buffer and chunk size	None	None
Removal of viewed files	UCF operation reads registry key	None

Table 86, page 409 describes the server configuration settings for UCF and HTTP modes.

Table 86. Server configuration settings in content transfer modes

Setting	UCF (server.config.xml)	HTTP
Temp working directory for upload/download	server.contentlocation*	server.contentlocation*
Tracing/debug	tracing.enabled	Limited: debug stops cleanup of temp files on server
Polling for config file change	file.poll.interval	None

Setting	UCF (server.config.xml)	HTTP
Log file location	DFC logging file in log4j.properties	wdk.log
File compression	compression.exclusion.formats	None

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 pop-up blockers interfere with content download because HTTP download opens a new window to display downloaded content. You can try turning off the pop-up 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 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 filename. 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 9-1. Getting multi-part upload files in the request

Files that are uploaded via HTTP are available from `HttpTransportManager`. `getUploadedFiles(paramName)`. The filename 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>
```

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 `wdk5download` 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
```

UCF overview

United Client Facilities (UCF) is a lightweight client-based service that transfers content between the client, application server, and Content Server. The UCF APIs provide a remote client presence that can access the client file system, registry, and network resources (URLs). On the application server, UCF provides the base client access to content library services such as import, export, checkin, and checkout.

UCF does not have built-in UI. It does not show any dialogs on its own. It provides APIs that a UCF client could use to display required UI.

Web-based applications running in browsers require a content transfer applet to establish the connection with the UCF server on the application server host. This applet must activate UCF on the client and pass the information required by each operation that is supported in the application.

Initializing content transfer controls

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

In the following example, the cancelcheckout component JSP page `cancelCheckout.jsp` has a radio control named `nodescendants`:

```
<dmf:radio name="nodescendants" group="group1" nlsid="
MSG_LEAVE_DESCENDENTS_CHECKEDOUT"/>
```

This control is initialized in the component definition as follows:

```
<init-controls>
  <control name="nodescendants" 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.

Configuring UCF components

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:

Table 87. UCF component definition elements

Element	Description
<code><ucfrequired></code>	Contains zero or more <code><events></code> and/or <code><pages></code> elements

Element	Description
<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.

Customizing a UCF component

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.

Refer to [Getting return values from content transfer, page 418](#) 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. For example, the `ViewProcessor` class implements `preProcess` to get the user preference for view format and then calls `super.preProcess` to continue 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)`. For example, the `LaunchEditor` class operates on files in the package:

```
public void preProcess(
    IServiceOperation op) throws ContentTransferException
{
    try
    {
        IDfOperation dfop = op.getDfOperation();
        if (dfop instanceof IDfCheckoutOperation)
        {
            //access the operation and checkout item
            //get item format and type
            //look up authoring application for format/type and launch
            ...}
    }
}
```

- `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. For example, the `CheckinProcessor` class implements `postProcess` to subscribe the object after it has been checked in. (It does not use the operation instance.)

```
public void postProcess(
    IServiceOperation op) throws ContentTransferException
{
    super.postProcess(op);
    try
    {
        doSubscribeObject();
    }
    ...
}
```

Troubleshooting UCF

The following topics provide troubleshooting steps and UCF logging instructions.

Problem scenario: "Initializing plug-in" page hangs — Troubleshooting steps:

1. See if UCF client is installed correctly, especially the configuration files. Look at paths in UCF client configuration file (refer to [Configuring UCF on the client, page 52](#)) to make sure user has permissions on the target directories.
2. Open the browser Java console look and for "invoked runtime: ... connected, uid: ..." Invoked runtime indicates successful launch of the process. A UID indicates successful connection to the UCF server.

UCF client is not responding — Troubleshooting steps:

1. For UCF timeouts, check whether anti-virus or personal firewall software on the application server is monitoring port 80, port 8080, or the application server port that is in use. You may need to turn off monitoring of the application server port.
2. Ensure that the user has a supported JRE version on the machine in order to initiate UCF installation. Supported JRE versions are listed in the DFC and Webtop application release notes. You can point the client browser to a Java tester utility such as [Javatester utility](#) to verify the presence and version of a JRE.
3. 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.
4. Are there any errors on the UCF server side? Check the application server console.
5. Restart the browser and retry the content transfer operation.
6. Kill the UCF launch process and retry the content transfer operation.
7. If UCF operations still do not launch, delete the client UCF folder located in `USER_HOME/username/Documentum/ucf`.

"Invalid checksum" error on client — The following error may be displayed:

Invalid checksum for the file "some_file_name."

This error is caused by cached applets that have a different checksum for UCF client files compared to those stored in the applet on the UCF server.

Workaround: Clear the cached applets in the client Java control panel and clean the browser cache.

Logging UCF

UCF server logging — 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`, which is located in the web application directory `WEB-INF/classes`. Add the following value to this file and restart the application server:

```
<option name="tracing.enabled">  
  <value>true</value>  
</option>
```

Note: The logging is verbose and can generate large logs in a short period of time.

UCF client logging — 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 `SEVERE`, `FINE`, `FINER`, or `FINEST`. For example:

```
handlers=java.util.logging.FileHandler, java.util.logging.ConsoleHandler
.level=FINEST
```

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)
```

Enable tracing to a log file in `ucf.client.config.xml`. Trace files (`ucf.trace*.log`) go to a specified location as shown in the example below:

```
<option name="logs.dir">
  <value>C:\Documentum\logs</value>
</option>
<option name="tracing.enabled">
  <value>true</value>
</option>
<option name="debug.mode">
  <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]
```

Getting return values from content transfer

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 88. 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 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.

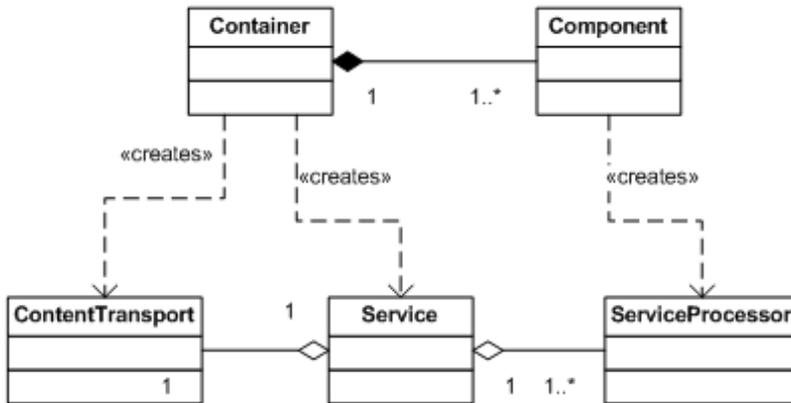
The container transfer container definition specifies a service class that extends `ContentTransferService` and acts as a controller for the UCF processor and transport classes as well as the WDK container class. The service sets up the DFC context (DFC session, operation) and reads the WDK container 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. [Figure 38, page 420](#) diagrams the relationship between the classes.

Figure 38. 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 [Customizing a UCF component, page 414](#)

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.

Displaying 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));
}

```

Logging and tracing content transfer component operations

WDK provides logging at the content transfer component level. WDK 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 `log4j.properties`. (The file `log4j.properties` is located in `WEB-INF/classes`.) By default, this file has the value of `log4j.appender.file.File=${user.dir}/documentum/logs/documentum.log`. This value is resolved to the subdirectory `/documentum/logs` under the application server executable directory, for example, `TOMCAT_HOME/bin/documentum/logs`.

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-
new session id added:
1;2284f8c1b0cfef3b1q113108b1q10338b260041q1d7ffe
...
[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
...
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  
...
```

Configuring and Customizing Roles and Client Capability

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, WDK defaults to using the client capability model with the following `client_capability` attributes which can be set on the `dm_user` object: `consumer`, `contributor`, `coordinator`, and `administrator` (refer to [Supporting roles and client capability, page 71](#)). 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-based actions, page 423](#)
- [Role-based filters, page 425](#)
- [Role-based UI, page 426](#)
- [Custom role plugin, page 426](#)
- [Role configuration overview, page 427](#)

Role-based actions

There are two ways you can base actions on roles:

- Actions can be performed only by certain roles

The following example restricts checkout of documents even though the user has VERSION permission. This is applied through a role precondition. In the 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"/>
  </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

Setting role precedence

When a user belongs to two or more roles that have different sets of available actions, either because the actions are scoped to roles or presets are applied to roles, the applicable scope or preset is randomly

applied. You must modify app.xml to specify the precedence of roles. The role precedence is applied based on the order in which roles appear within the <rolesprecedence> element. For example, if the user is a member of both the consumer and the administrator roles, and the consumer role is listed first, the user will not have administrator presets applied.

The following example in a file named app_modifications.xml in custom/config, sets the administrator role to take precedence over the consumer role in repository A and the opposite precedence in staging repository B:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<config><scope>
<application modifies="wdk/app.xml">
<insert path=rolemodel>
  <rolesprecedence>
    <filter doctype="A">
      <role>administrator</role>
      <role>coordinator</role>
      <role>contributor</role>
      <role>consumer</role>
    </filter>
    <filter doctype="B">
      <role>consumer</role>
      <role>contributor</role>
      <role>coordinator</role>
      <role>administrator</role>
    </filter>
  </rolesprecedence></insert></application></scope></config>
```

When you configure role precedence in app.xml and refresh the configurations in memory, the precedence may not be applied for up to ten minutes, because roles are cached for ten minutes on the application server. You can stop and restart the application server to get an immediate change in precedence.

Note: After a filtered precedence is in memory, it cannot be refreshed by the configuration service. (No filtered definitions can be refreshed.) You must stop and restart the application server to get a change in filtered definitions.

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>
```

Custom role plugin

WDK provides two alternative role model plugins to support user roles:

- Client capability role model
- Repository 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 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 423](#) for more information.

- Role-based filters

You can restrict a container so 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 425](#).

- 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 426](#) for more information.

The role service gets the user's role using either the repository role model plugin (default, supporting role groups in the Content Server), the client capability plugin (refer to [Supporting roles and client capability, page 71](#)), or both. [Table 89, page 428](#) describes two different ways role can configure action and component behavior.

Table 89. Uses of role in configuration

Configuration file location	Effect
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

Customizing the Application Framework

The following topics provide information on the application framework and customization points provided by the framework:

- [Application elements interaction, page 429](#)
- [Configuration service overview, page 430](#)
- [Improving performance, page 437](#)
- [Tracing, page 446](#)
- [Logging, page 448](#)
- [Testing components, page 449](#)
- [Utilities, page 449](#)
- [Using the comment stripper utility, page 452](#)
- [Making an application accessible, page 453](#)
- [Troubleshooting runtime errors, page 455](#)

Application elements interaction

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 service overview

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. The application server console and log will display runtime errors due to incorrect XML values in configuration files including 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 Java EE servers. You must refresh the definitions in memory 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

The configuration service is supported by the following classes:

- [ConfigService \(ConfigService, page 431\)](#)
Provides access to config file settings
- [IConfigLookup \(Configuration lookup, page 432\)](#)
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 \(IConfigContext, page 431\)](#)
Instance of the configuration service that can be used to determine whether a specified context name matches a defined qualifier name
- [IConfigReader \(Configuration reader, page 434\)](#)
Loads and reads configuration files
- [IConfigLookupHook \(Configuration lookup hooks, page 433\)](#)
Allows configuration lookup calls to be interrupted and overridden
- [IQualifier \(Scope, qualifier, and Context classes, page 46\)](#)
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 qualifer by calling isContextName().

Example 11-1. 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 configuration lookup methods `lookupString`, `lookupInteger`, and `lookupBoolean` have an optional parameter `consultPreference`. Set to `false` to look up a configuration value from the component

definition and bypass a lookup of the user preference when the lookup is not needed. This will enhance performance.

The `getContext()` method returns the component's current context.

Example 11-2. 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 11-3. 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 `Environment.properties` file which is located in `WEB-INF/classes/com/documentum/web/formext`. 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 11-4. 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 Java EE 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 Java EE 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
```

Context

The `Context` object holds 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 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 it can be passed as a URL argument or string. To serialize the context, use `Context.serialize()`.

Example 11-5. 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 = URLEncoderCache.getEncodedString(
    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 11-6. Saving an object ID in context

The `DocList` component class saves the object ID and object type in a `Context` object::

```
context.set("objectId", DfId.DF_NULLID_STR);
context.set("type", "dm_docbase");
```

When the user selects a document in the drill-down view for viewing, the `onClickObject()` method of the `DocList` class gets the `Context` to pass to the view action:

```
ActionService.execute("view", args, getContext(), this, null);
```

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 11-7. 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 processing

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 437)

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.

Improving performance

There are several application guidelines that can significantly improve performance of your web application. These interventions are described in the following topics.

Follow these 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

By default, arguments in multiple selection are passed in a query string. One query string is created for selected object. Alternatively, you can cache arguments in the container class. For information, refer to [Caching component arguments for multiple selection, page 159](#)

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 10 selectable items and 100 associated actions, 1000 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.

You can configure the application to test action preconditions only when they are executed instead of on page rendering. Set the `onexecutiononly` attribute of the precondition element to true as follows:

```
<precondition onexecutiononly="true" class=.../>
```

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 [Adding custom attributes to a datagrid, page 210](#) 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 [Using the comment stripper utility, page 452](#) for information on using this tool.

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.

Controls can retrieve any number of rows from a data provider unless you limit the cache size or apply paging to the datagrid. The memory cache continues to grow as the user pages through entries, because all attributes for displayed columns are cached in memory. An optimization setting will limit the caching to object IDs only.

The 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`. This value defaults to 100, which will cache the values for page sizes up to 100. If you increase the available page size in your application, you should increase the cache size to match the largest page size. Paging is configured on a JSP page that contains a datagrid. Limit the choices for page sizes by setting the `pagesizevalues` of the `datapagesize` JSP tag.

The cache optimization setting `useOptimizedResultCache` in the properties file `Databound.properties` located in `WEB-INF/classes/com/documentum/web/form/control/databound` limits caching to object IDs only. This value is set to `true` by default, and object IDs are cached and data rows are

retrieved only for the current page in a listing display. An optimized cache is used for the Cabinet, HomeCabinet, and MyFiles components. If your listing component extends `objectlist`, `myfiles_classic`, or `homecabinet_classic`, you will inherit the optimization support.

To add optimization support to a listing component, you must construct an alternative, simplified query that does not query all of the display attributes. Pass your query to the `DocbaseQueryService` method `buildObjectListFindByIDQuery()`. Refer to the source code for `DocList` in `webcomponent/src/com/documentum/webcomponent/navigation/doclist` for a detailed example.

Java EE memory allocation

If the memory allocated to the Java EE 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 Java EE server start script that you use to start your application server:

```
1 -Xms512m 2-Xmx512m 3-verbose:gc
```

- 1 Starting memory heap size, in megabytes. 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. 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

Monitor memory usage by the Java process in the Windows task manager to determine whether your memory allocations are optimum. 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

Java EE 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 "Connection 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.

For more information on these settings, refer to your application server documentation.

HTTP sessions

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 Java EE session timeout when the top browser frame is unloaded, such as when the user navigates to another website. 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.

Preferences

User preferences are stored as cookies and written to the repository. Since cookies are passed back and forth with every request and response, there is a small increase in network traffic.

The configuration lookup methods `lookupString`, `lookupInteger`, and `lookupBoolean` have an optional parameter `consultPreference`. Set to `false` to look up a configuration value from the component definition and bypass a lookup of the user preference when the lookup is not needed.

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 3. If the value is empty or zero, then history is maintained indefinitely. This setting could significantly affect performance. Decrease the memory footprint per user by setting this value lower. If you set it higher, it will consume more memory.

Too many form history objects can use up memory. Set the upper limit for the number of objects as the value of `maxNoOfFormHistoriesThreshold` in `FormProcessorProp.properties`. The default value is 50. A message will be displayed if the user tries to navigate past the maximum number of pages in history.

Memory that is allocated to maintaining browser history is managed more efficiently on the Java EE server if you generate framesets and frames using the `<dmf:frameset>` and `<dmf:frame>` tags. Refer to [Managing frames, page 176](#) 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. Do the following to enhance this performance:

- For each value assistance query, use Documentum Application Builder to turn on the option to allow caching.
- Turn on client persistent caching in `dfc.properties`, which is located in `WEB-INF/classes`:

```
dfc.cache.enable_persistence = T
```
- Index the associated attributes in Content Server.

Search query performance

Set `<displayresultspath>` to `false` in your custom search component definition to speed all queries. This suppresses the query for folder path of each object.

In advanced search, you can add a checkbox for case-sensitive search for non-indexed repositories. 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 AST indexer is not case-sensitive, so the case-sensitive checkboxes are applicable only to non-indexed 5.3 Content Server.

High latency and low bandwidth connections

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-` 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-` 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>
```

For better performance, your qualifier should implement the `IInquisitiveQualifier` interface. At startup, this interface is used to inform the qualifier of all relevant scopes defined in the action and component definitions. The qualifier can return an empty scope value that is cached, when the runtime context is not relevant.

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 proxy servers, page 63](#) 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 and performance

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 303](#) for more information.

Tracing

You can use the WDK tracing flags in your JSP pages and Java classes and 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. By default, this directory is located in the executables directory of the application server, for example, `C:\tomcat6_0_10\bin\documentum\config`.

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. For a description of each WDK tracing flag and the operations that it traces, refer to [Appendix A, WDK Tracing Flags](#).

Turning on WDK tracing

To enable tracing for the current session, navigate to `wdk/tracing.jsp` and check the box that enables tracing. 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`.

Using DFC tracing

To turn on DFC tracing, set the following preference in your `dfc.properties` file located in `WEB-INF/classes`:

```
dfc.diagnostics.resources.enable=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.

Adding custom tracing flags

You can add tracing flags specific to your application, and then use the flags in your application code.

Example 11-8. 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.

```
String timeoutValue = request.getParameter(TIMEOUT_PARAM);
if (Trace.MYTIMEOUTCONTROL)
{
    Trace.println("TIMEOUT_PARAM value read: " + timeoutValue);
}
```

Using 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 pop-up 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 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 contained in the WAR file in `WEB-INF/lib`. 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 in `WEB-INF/classes`. The log filename (not path) is specified in `log4j.properties` as the value of the key `log4j.appender.file.File`:

```
${user.dir}/documentum/logs/documentum.log${user.dir}/documentum/logs/documentum.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 require logging. 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 website](#).

Turn on `log4j` logging for individual DFC classes by adding a line to `log4j.properties` located in `WEB-INF/classes`. The following example adds debugging for query builder queries:

```
log4j.logger.com.documentum.fc.client.search.impl.broker.
  DocbaseBroker=DEBUG
```

You can enable logging of DFC tracing with the `log4j` package for DFC classes in the file `dfc.properties`, which is located in `WEB-INF/classes`. By default, DFC logging is suppressed. You can turn on tracing of method calls, parameters, return values, trace formatting, and tracing depth. Replace this with the full path and filename. To enable logging of all DFC trace calls, change the following key to true:

```
dfc.tracing.enable=true
```

The DFC trace file `dfctrace.log` is written to the Java EE server directory that contains the app server executable. For example, in Tomcat the trace file will be located in the `bin` directory.

Testing components

WDK provides a sample library of JSP pages and server classes that test and display all of the WDK controls. This library is provided as an archive on the download site. Unzip the archive into your WDK-based application to use the samples.

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`.

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 and Webtop Reference*.

Utilities

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 com.documentum.web.common.URLEncoderCache;
...
bufferUrl.append("&");
bufferUrl.append(strArgName).append("=");
if (strArgValue != null && strArgValue.length() >0)
{
    bufferUrl.append(URLEncoderCache.getEncodedString(
        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();
}
```

Version — 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;
}
```

Using the comment stripper utility

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`. [Table 90, page 452](#) describes the parameters to use in starting this tool from the console.

Table 90. Comment stripper utility parameters

Parameter	Description
<code>args filename</code>	Removes comments from a single file
<code>args *.ext</code>	Removes comments from all files with the specified extension
<code>?</code>	Displays help
<code>l</code>	Removes leading white space
<code>t</code>	Removes trailing white space
<code>m</code>	Removes HTML comment blocks <code><!--...--></code> and <code><!--...--></code>
<code>j</code>	Removes JSP and JavaScript <code>/* ... */</code> comments
<code>r</code>	Recurses directories from current
<code>oxx</code>	Uses specified extension instead of overwriting original file
<code>v</code>	Outputs in verbose mode (OFF by default)

Making an application accessible

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 turning on accessibility by default, refer to [Enabling accessibility \(Section 508\), page 70](#).

Accessibility of standard HTML tags is not described here. Information is available on the Internet for creating accessible HTML markup.

The following topics describe WDK accessibility features and how to use them to create an accessible web application.

Making labels accessible

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>
```

Providing image descriptions

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

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 this 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.

Adding 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 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"/>
```

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 15, page 71](#) for details.
- Alt text strings can be configured for images. Refer to [Providing image descriptions, page 453](#) 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.

Troubleshooting 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 when you make changes to them. Visit `wdk/refresh.jsp` to refresh the configurations in memory.

Error loading main component

The following error appears when the WDK-based application is first loaded:

```
This <main> component is invoked when the root application
is loaded and when timeout and release operations execute.
Customize this component to load the application start page.
```

Cause: Your `custom\app.xml` extends `webcomponent\app.xml`.

Solution: 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. Object type information is cached on the application server host.

Workaround: Delete the cache. The cache directory is located in a documentum subdirectory of the application server instance root directory. 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.

JavaScript error on application connection

When you log into 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 using IE without a Java virtual machine. You must upgrade to a supported version of the browser and install a supported VM.

"Configuration base has not been established"

This error is often 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() :
```

Solution: Verify the paths in your action or component definition to make sure they are correct. 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.)

Application slows down

The application server can slow down even when the system has a lot of memory available.

Solution: Increase the Java memory heap. The maximum memory size for the JVM should be set to 1024. Refer to *Web Development Kit and Webtop Deployment Guide* for more information.

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**.

Application runs out of sessions

There are several reasons that can cause an application to run out of sessions:

- The Content Server has run out of sessions. For information on configuring sessions for the Content Server, refer to *Content Server Administration Guide*.
- The WDK-based application's `dfc.properties` file does not provide enough sessions. The value of the key `dfc.session.max_count` should be at least 1000.
- Session pooling is turned off. If session pooling is turned on, the DFC 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>
```

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.

Solution: Use UCF transfer.

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.

Solution: Encode XML files in UTF-8, or use the Sun Java plugin for all users. Do not use Notepad to edit an XML file.

3. If the XML application DTD does not have a .dtd extension, a null pointer exception is displayed.

Solution: Rename the DTD file with a .dtd extension.

Unable to locate checked out objects after deploying a WDK-based application

If your client hosts have checked out objects using DFC 4.2 or lower, they will not be located by DFC 6. Documentum began using the HKEY_CURRENT_USER registry hive recommended by Microsoft with version 4.3 of Documentum products.

Solution: If your clients have objects still checked out when they connect to a WDK 6 client application, they can manually check in those objects by selecting **Checkin from File** in the checkin component UI. It is recommended, however, that clients check in all objects before connecting to a WDK 6 client application.

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.

Solution: Make sure that the component whose JSP page contains the control is getting a DFC session.

Tags all have the same label

Application servers have a setting that reuses JSP tags (tag pooling). Refer to *Web Development Kit and Webtop Deployment Guide* for instructions on disabling tag pooling.

Custom authentication

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.

Persession 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.

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

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 similar to the following:

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

The following topics describe special login situations. Common authentication configuration is described in the Administration chapter. Refer to [Enabling login and session options, page 63](#)

Using 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. Spaces must be escaped (replaced with `%20`).

```
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, 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 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.

Note: With ticketed login, both its creation time and expiration time are recorded as UTC time. This ensures that problems do not arise from tickets used across time zones. When a ticket is sent to a server other than the server that generated the ticket, the receiving server tolerates up to a three minute difference in time to allow for minor differences in machine clock time across host machines. System administrators must ensure that the machine clocks on host machines on which applications and repositories reside are set as closely as possible to the correct time.

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 `URLEncoderCache` in the package `com.documentum.web.common`.)

```
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 + URLEncoderCache.getEncodedString(
        ticket) + strSuffix;
    System.out.println(fullUrl);
}
finally
{
    if(sess != null)
    {
        releaseSession(sess);
    }
}
```

Skipping authentication for a component

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, 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 the login dialog is not called.

Configuring and Customizing Webtop

The entry to the Webtop application is through either `index.html` or `default.html` in the root web application directory. Each HTML file has the same content, which redirects to the component dispatcher servlet with the URL for the "main" component.

Webtop configuration and customization is described in the following topics:

- [Webtop directory structure, page 465](#)
- [Webtop views, page 465](#)
- [Using Webtop qualifiers, page 467](#)
- [Interaction of inbox and task manager components, page 468](#)
- [Creating bookmark URLs \(favorites\), page 471](#)
- [Redirecting to Webtop, page 472](#)
- [Customizing the Webtop browser tree, page 472](#)
- [Using Webtop navigation utilities, page 476](#)

Webtop directory structure

The top application layer that contains content in Webtop is the `webtop` directory. The custom application layer extends the webtop application definition and becomes the top application layer. The custom directory should contain your custom configuration files, NLS resource files, and JSP pages. Place your custom classes under the directory `WEB-INF/classes/com/documentum/custom`.

Webtop views

The main component definition (`webtop/config/mainex_component.xml`) specifies the start page as `webtop/main/mainex.jsp`. Webtop 5.3.x had two views, `classic` and `streamline`. The `streamline` view is turned off by default in version 6. To enable the `streamline` line in your custom `app.xml` file, set the value of `<streamlineviewvisible>` to `true`.

Figure 39, page 466 shows the default entry page in Webtop. The index page at the root of the Webtop application loads the main component, displayed by mainex.jsp.

Figure 39. Main view in Webtop

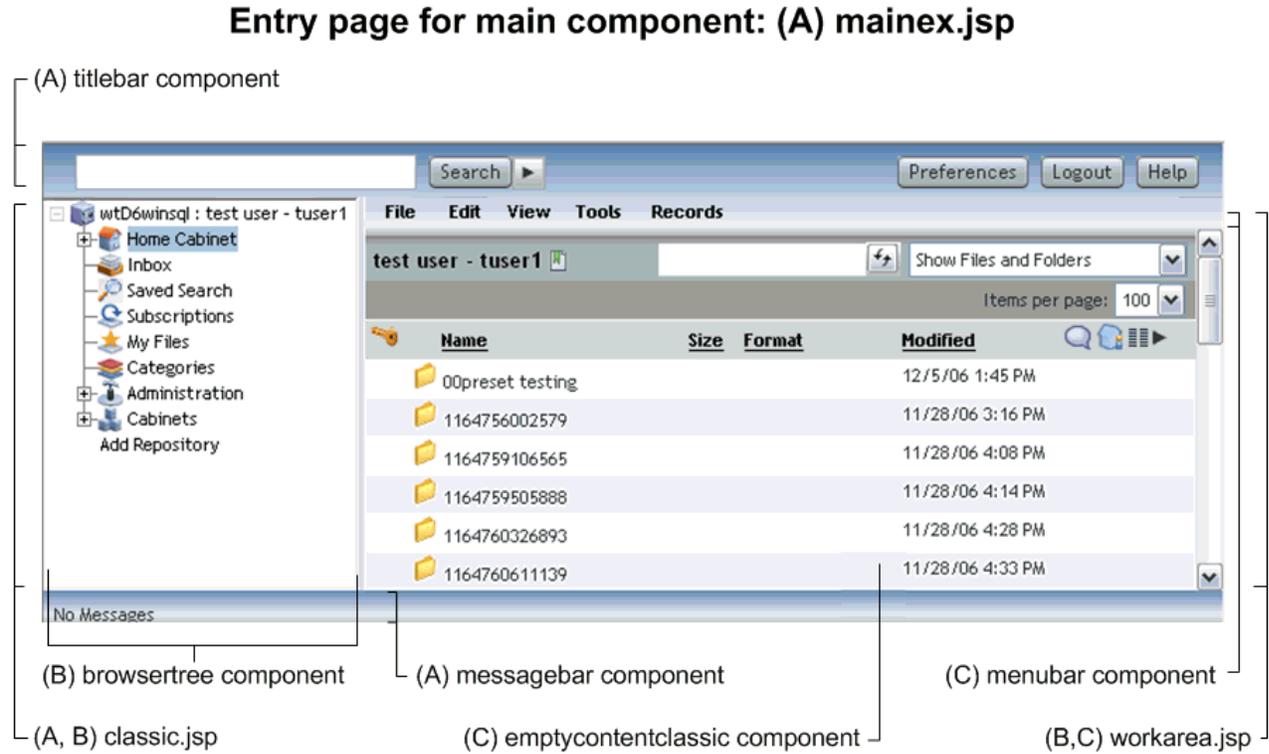


Table 91, page 466 describes the components that are displayed in this entry UI.

Table 91. Components and pages that are loaded by the main Webtop component

JSP page	Frames
(A) mainex.jsp	(A) timeout control (hidden frame)
	(A) titlebar component
	(A) messagebar component
	(A, B) classic.jsp
(B) classic.jsp	(B) browsertree component
	(B, C) workarea.jsp

JSP page	Frames
(C) workarea.jsp	menubar component
	emptycontentclassic component (content frame into which browsertree or menu selections are loaded)

Note: The toolbar component from 5.3.x is hidden in version 6 because all of its functionality is available in right-click menus. To re-enable the toolbar in the JSP page classic.jsp, change the following line:

```
String strRows = "0,*"?
```

To this:

```
String strRows = "22,*"?
```

The streamline view is maintained for compatibility with 5.3 customizations.

Using Webtop qualifiers

Webtop adds a location qualifier that you can use to scope an action or component. The qualifier class `ApplicationLocationQualifier` matches the context value "location" to the scope "location". This feature allows you to present a different set of menu items depending on the user's location in the tree. The location specifies a global indicator of the current context of the application, which is either the current content component (e.g. doclist, subscriptions) and a custom path understood by that component, or the current doctype folder path. The configuration service adds the current repository name to the beginning of the path. The syntax for location scope is:

```
<scope location="/cabinet_or_folder_name/etc" >
```

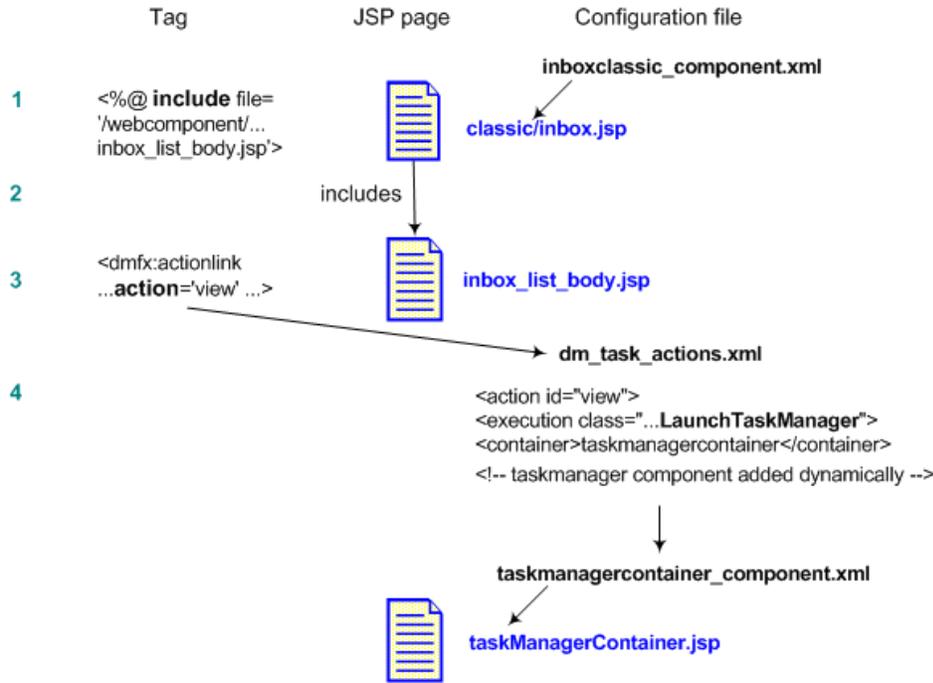
In the following example of an action definition, a special menu is defined for subnode 1 in the tree. The specified action will be available only when the user is looking at subnode 1, and the remainder of the nodes in the tree share a common menu:

```
<config>
  <scope location="mytree_node_componentID/subnode1">
    <action id="myaction">... .. </action>
  </scope>
  <scope>
    <action id="newdocument"> ... </scope>
</config>
```

Interaction of inbox and task manager components

The group of components that display a user’s inbox and task manager is large and complex in its interactions. The first sequence of interactions between components and actions is described in this section. [Figure 40, page 468](#) illustrates the sequence of processing in the inbox component.

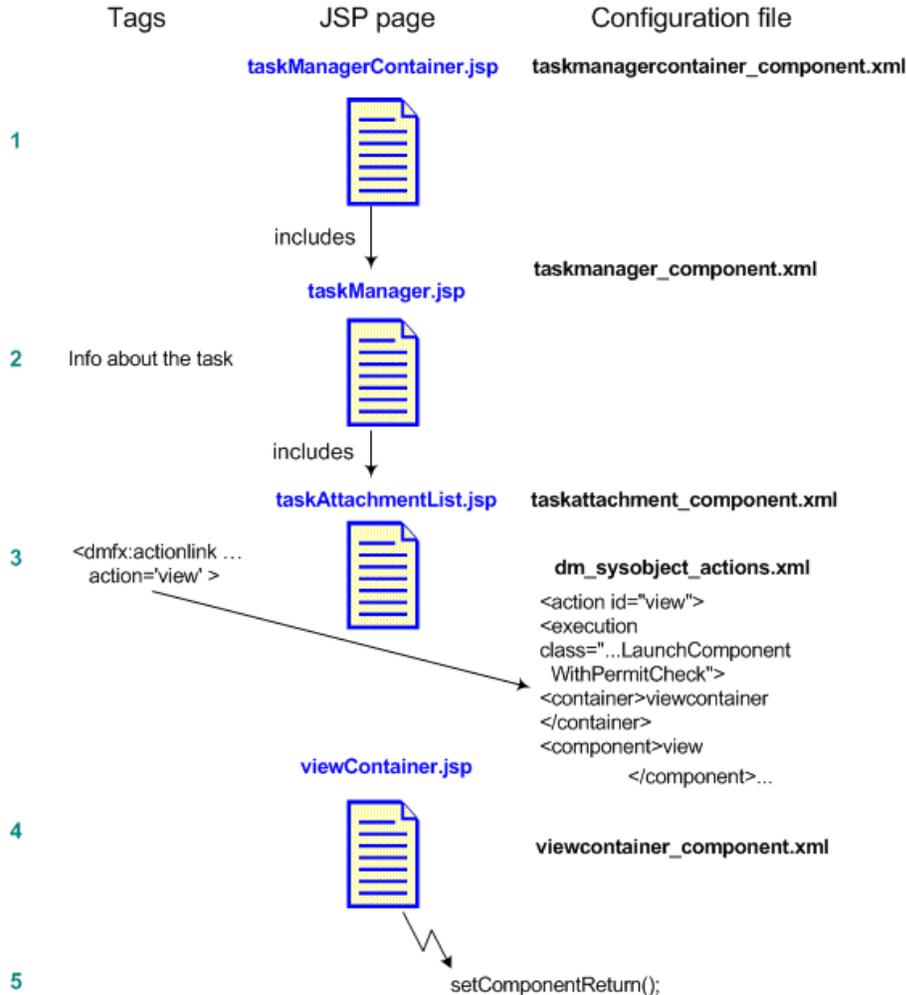
Figure 40. Inbox processing sequence



1. The inbox component renders the JSP page `inbox.jsp` in Webtop.
2. The Webtop UI pages includes `inbox_list_body.jsp` from the webcomponent layer.
3. The page `inbox_list_body.jsp` contains an `actionlink` control that calls the `view` action.
4. The `view` action uses the `LaunchTaskManager` class to launch the `taskmanagercontainer` component.

[Figure 41, page 469](#) diagrams the process in the task manager. The user selects a task and views the task attachment.

Figure 41. Viewing a task in the task manager

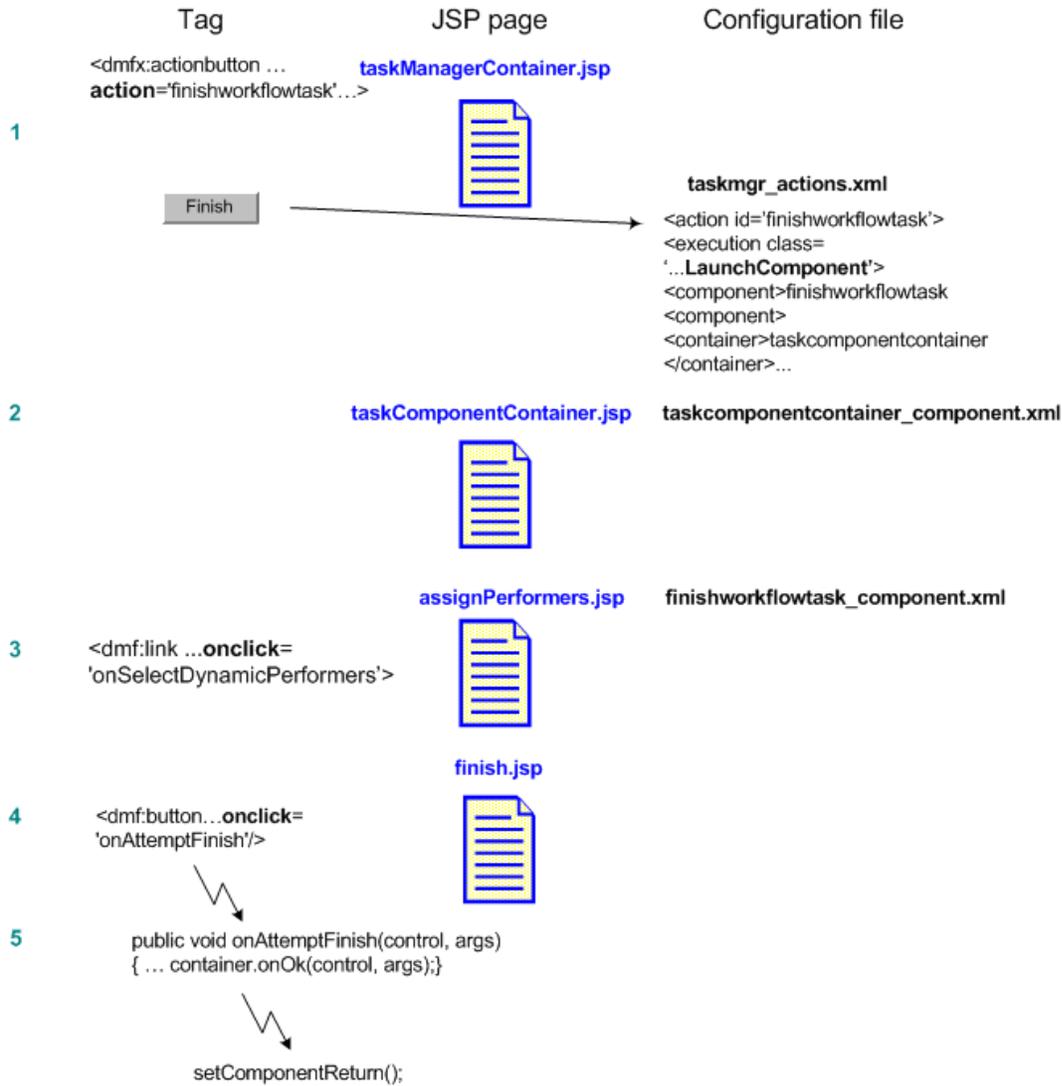


1. The `taskmanagercontainer` component is called by the view action in the inbox component. The container page `taskManagerContainer.jsp` includes the components listed in the `taskmanagercontainer` definition. The first component to be displayed is the `taskmanager` component, on the **Info** tab.
2. The `taskmanager` component UI displays information about the selected task and includes the component `taskattachment`.
3. The `taskattachment` component UI displays a list of attachments for the task and a link to view one or more selected attachments.
4. When the user selects the view link for an attachment, the view action launches the view component in the view container.

- After the selected attachments have been viewed, the ViewContainer class calls setComponentReturn, which returns to the calling component taskattachment.

Figure 42, page 470 describes the finish task process.

Figure 42. Finishing a task



- The taskmanagercontainer has several buttons that perform actions on the workflow. For example, the **Finish** button launches the finishworkflowtask action.
- The finishworkflowtask action launches thefinishworkflowtask component in the taskcomponentcontainer.

3. The finishworkflowtask component starts with an assignPerformers page. After the user assigns performers, the component class calls the finish page.
4. The finish page contains a signoff field that contains a hidden button. The button onclick attribute specifies the onAttemptFinish() event handler.
5. The component class onAttemptFinish() event handler calls the container onOk() method, which returns to the calling component (inbox).

Creating bookmark URLs (favorites)

Webtop is launched via a URL such as `webtop/component/main`. You can add URL arguments that direct the user to a specific location.

The URL has the following arguments (user-defined values are shown in italics):

```
/Web_application/component/main?entryPage=  
page_name&entrySection=section_name&objectId=object_ID
```

The user-defined values are described below.

entryPage — Specifies the type of view to start in. Supported values: classic and streamline. If entryPage is not specified, the user's preference is used. If the user has not set a preference, the default is set in the main component definition, in the configuration file `main_component.xml`.

entrySection — Determines the browsertree node (classic view) or tab (streamline view) to start in. If entrySection is not specified, the cabinets view is displayed by default. The section values are described below:

- *Classic view*

The section is the Docbase tree node. Valid values are `homecabinet_classic`, `cabinets`, `inboxclassic`, `subscriptions_classic`, `myfiles_classic`, and `administration`. These values correspond to node IDs in the browsertree component definition.

- *Streamline view*

(Deprecated) The section determines the start tab.

objectId — Determines the object of the bookmark (a folder or an object). If the ID is a folder or cabinet, the folder or cabinet is displayed. If the ID is an object, the contents of the containing folder are displayed. The objectId argument cannot be used with an entrySection argument.

If a login is required for the user who launches a bookmark, the login dialog is presented.

Sample URLs to Webtop views — With all forms of URL addressing, Webtop launches a login dialog if required. If an object ID is specified, the appropriate repository is selected in the login dialog. Sample URLs include the following:

- URL to classic view, using preferences for the selected tree node:
`/webtop/component/main`
- URL to classic view, with myFiles component node as starting point:
`/webtop/component/main?entrySection=myfiles_classic`
- URL to a folder "xyz" (substitute actual folder ID):
`/webtop/component/main?objectId=xyz_folderID`

Redirecting to Webtop

The Webtop frameset is designed to be the top frameset in a web application. If you are customizing Webtop, you must include the Webtop framework with your application components, and your components must be defined within the Webtop framework. Alternatively, add a JavaScript frame-breaker to the default.html and index.html pages of Webtop.

Example 12-1. Redirecting to Webtop from Another Application

The following example adds the highlighted lines to the redirect function in default.html::

```
<script>
function redirect()
{
    if (top.location != location)
    {
        top.location.href = document.location.href ;
    }
    var strPath = window.location.pathname;
    var nIndex1 = strPath.indexOf("/");
    var nIndex2 = strPath.indexOf("/", nIndex1 + 1);
    var strVirtualDir = strPath.substring(
        nIndex1 + 1, nIndex2);
    var dmfUrl = "/" + strVirtualDir +
        "/component/main?Reload=" + new Date().getTime()
        + "&__dmfUrl=" + dmfUrl;
}
</script>
```

Customizing the Webtop browser tree

Each node in the browsertree has an icon and label to help identify the node. The browsertree component has a set of client event handlers to update the current folder path shown in the object list component based on the user's position in the browser tree. The tree can also be updated from client events fired in the object list component.

Each node on the tree represents a container, such as a cabinet or folder. When the user selects a tree node, the tree refreshes and fires the client event `onNodeSelected`, with an argument of the folder or cabinet ID. The object list event handler for `onNodeSelected`, in `classic.jsp`, posts a component jump to the object list component.

Add custom nodes to the tree by copying the browser tree XML configuration file and overriding its component definition. The custom node will be rendered once for each available Docbase. There are three types of custom nodes:

- Non-Docbase node

When selected, this node will launch a custom component that does not require a Docbase session, similar to the Preferences component.

- Docbase node with no child nodes

This node will be rendered once for each available Docbase in the tree. When selected, this node will launch a custom component that is Docbase-related, similar to the Inbox component.

- Docbase-related node with child nodes

This node will be rendered once for each available Docbase node in the tree. When selected, this node will launch a custom component that requires a Docbase session and can have child nodes, similar to the subscriptions component.

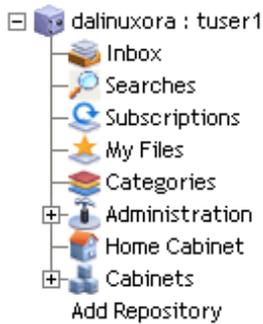
Add a `<node>` element to the browser tree component definition, with a `componentid` attribute whose value is the ID of your custom component. For example:

```
<nodes>
  <node componentid='my_component'>
    <icon>my_icon.gif</icon>
    <label>My Component</label>
    <handlerclass>com.acme.MyComponentNode</handlerclass>
  </node>
</nodes>
```

To support child nodes, you must have a handler class that extends `com.documentum.webtop.control.BrowserNavigationNode`.

Example 12-2. Removing the Add Repository Option from the browser tree

To restrict users from adding repositories, create simple extensions of the Webtop `browsertree` class and tag class and the `browsertree` component class to do this. The following screen shows the **Add Repository** option at the bottom of the browser tree:

Figure 43. Browser tree Add Repository option

Create your custom control class as follows. When the control initializes, it will display only the nodes that are defined in the browsertree component and set the visibility of the dynamic nodes to false:

```
package com.mycompany;
import java.util.Iterator;
import com.documentum.web.common.ArgumentList;
import com.documentum.web.form.control.TreeNode;

public class WebTopBrowserTree extends com.documentum.webtop.control.
WebTopBrowserTree
{
    public void onInit(ArgumentList args)
    {
        super.onInit(args);
        Iterator it = getAllNodes();
        if (null != it)
        {
            Object m_Node = null;
            while (it.hasNext())
            {
                m_Node = it.next();
            }
            if (null != m_Node)
            {
                ((TreeNode) m_Node).setVisible(false);
            }
        }
    }
}
```

You must create a tag class to get the new control:

```
package com.mycompany;
public class WebTopBrowserTreeTag extends com.documentum.webtop.control.
WebTopBrowserTreeTag
{
    protected Class getControlClass()
    {
        return com.mycompany.WebTopBrowserTree.class;
    }
}
```

Now create a tag library file `custom.tld` in `WEB-INF/tlds`. Copy the content of `dmwebtop_1_0.tld` and change only the `shortname` element for the tag library and the tag class for the `browsertree` control, as shown:

```
...
<shortname>ct</shortname>
<tag>
  <name>browsertree</name>
  <tagclass>com.mycompany.WebTopBrowserTreeTag</tagclass>
  ...
</tag>...
```

Create a modification XML file to modify the Webtop `browsertree` component definition to use your custom start page and custom class. Save this file in `custom/config`:

```
<config version="1.0">
<scope>
  <component modifies="browsertree:
    webtop/config/browsertree_component.xml">
    <replace path="pages.start">
      <start>/custom/browsertree/browsertree.jsp</start>
    </replace>
    <replace path="class">
      <class>com.mycompany.BrowserTree</class>
    </replace>
  </component>
</scope>
</config>
```

For the last step, copy the Webtop `browsertree` JSP page `webtop/classic/browsertree.jsp` to `custom/browsertree`. Specify your custom tag library:

```
<%@ taglib uri="/WEB-INF/tlds/custom.tld" prefix="ct" %>
```

Import your custom control class by importing it in place of the Webtop control in the import statement:

```
<%@ page import="com.documentum.web.form.Form,
  com.documentum.web.common.ClientInfo,
  com.documentum.web.common.ClientInfoService,
  com.mycompany.BrowserTree" %>
```

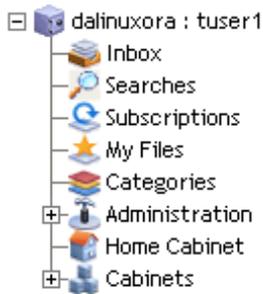
Search for the `browsertree` tag:

```
<dmwt:browsertree name='<%=BrowserTree.TREE_CONTROL%>' .../>
```

Replace it with your custom tag:

```
<ct:browsertree name='<%=BrowserTree.TREE_CONTROL%>' .../>
```

Stop and restart the app server to pick up your custom classes and verify that the **Add Repository** option is not visible:

Figure 44. Add Repository option is hidden

You must also remove the **Repositories** tab from the preferences component. Modify the preferences component definition in `webcomponent/config/environment/general` as follows:

```
<component modifies="preferences:
  webcomponent/config/environment/preferences/preferences_component.xml">
```

Add the following modification, which will display only the components that you list here:

```
<replace path="contains">
  <contains>
    <component>general_preferences</component>
    <component>display_preferences</component>
    <component>vdm_preferences</component>
    <component>savecredential</component>
    <component>search_preferences</component>
    <component>format_preferences_summary</component>
  </contains>
</replace>
```

Refresh the app server by navigating to `wdk/refresh.jsp` and then view the preferences component to verify that the **Repositories** tab is removed:

Figure 45. Repositories tab removed

Using Webtop navigation utilities

Webtop provides application navigation utilities. The `AppSessionContext` class gets and sets the type of view and application location.

The `AppSessionContext` class stores context settings in the HTTP session and adds tracing. This class also has a getter method for `ApplicationLocation`. To get the current folder path in your component, call `AppSessionContext.getAppLocation()`.

Example 12-3. Getting the application location

The location specifies a global indicator of the current context of the application. The current context is either of the following, prepended with the current Docbase name:

- The current content component, such as doc list, and a custom path understood by that component, in the form *componentid/custompath/custompath...*
- The current Docbase folder path, in the form *docbasename/folderpath/folderpath...*

In the following example of a component `onInit()` method, `AppSessionContext` provides the HTTP session and the application location (in the form of the `ApplicationLocation` object):

```
public void onInit(ArgumentList args)
{
    // overwrite the initial folder path from the app context if it's not set
    if (args.get(PARAM_FOLDERID) == null &&
        args.get(PARAM_FOLDERPATH) == null)
    {
        AppSessionContext context = AppSessionContext.get(getPageContext().
            getSession());
        args.replace(PARAM_FOLDERPATH, context.getAppLocation().
            getFolderPath());
    }
    super.onInit(args);
}
```


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. Tracing flags in `com.documentum.web.test` are not supported.

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.

Enable tracing for all sessions for setting `SESSIONENABLEDBYDEFAULT` to true in `WEB-INF/classes/com/documentum/debug/TraceProp.properties`.

WDK tracing flags are described in the following topics.

Tracing sessions

[Table 92, page 479](#) describes the tracing flags in `com.documentum.web.common.Trace` and `web.formext.Trace` that can be used to trace HTTP and Docbase sessions:

Table 92. Session tracing flags

Flag name	Description
<code>SESSIONSTATE</code>	Traces changes to HTTP session object and attributes
<code>SESSIONSYNC</code>	Traces session locking and unlocking
<code>SESSIONENABLEDBYDEFAULT</code>	Traces all sessions when set to true
<code>SESSIONTIMEOUTCONTROL</code>	Traces changes to the HTTP session timeout defaults through the <code>SessionTimeoutControl</code> servlet

Flag name	Description
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 ReplicatedSession and Container.isFailoverEnabled(). 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.
CLIENTSESSIONSTATE	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

[Table 93, page 480](#) describes the tracing flags in `com.documentum.web.common.Trace` and `web.formext.Trace` that can be used to trace framework operations.

Table 93. Framework tracing flags

Flag name	Description
THREADLOCALVARIABLE	Traces binding of variables to threads.
LOCALESERVICE	Traces set locale and create locale hook

Flag name	Description
ACTIONSERVICE	Traces getActionDef(), queryExecute(), preconditions and results, execute(), return values
CONFIGSERVICE	Traces lookupElement(), inheritance resolution, qualifiers, lookup hooks, IConfigElement()
PREFERENCES	Traces preference setting, caching, and retrieving
ROLESERVICE	Traces role preconditions called by action service
BRANDINGSERVICE, CONFIGTHEMERE-SOLVER	BRANDINGSERVICE traces the resource folder used for branding; CONFIGTHEMERE-SOLVER 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
PERSISTENTOBJECTCACHE	Traces caching and retrieval of persistent objects such as DFC objects
CONFIGMODIFICATION	Traces usage of the modification mechanism including the type of modification (insert, remove, replace) and modification target element
PERMISSIONS	Traces attempts to revoke basic permissions and to grant or revoke extended permissions

Tracing controls and validation

Table 94, page 481 describes the flags in com.documentum.web.form.Trace and com.documentum.web.common.Trace that can be used to trace controls and validation.

Table 94. Control and validation tracing flags

Flag name	Description
CONTROL	Traces the TreeTag renderEnd event

Flag name	Description
CONTROLTAG	Traces image folder or file path resolution error, control creation,doStartTag and doEndTag events
AUTOCOMPLETESERVICE	Traces calls to the service to get the user's autocompletion list, adding to it, and clearing it
BREADCRUMBSERVICE	Traces jump to component on breadcrumb; change of mode from relative to absolute; browsertree refresh; jump to root component; update state
HOTKEYSSERVICE	Traces lookup of hotkey ID and NLS ID in config file and key combination in NLS file
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

Table 95, page 483 describes the flags in com.documentum.web.form.Trace that can be used to trace form (JSP page) processing.

Table 95. Form and page tracing flags

Flag name	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);
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.

Flag name	Description
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).
FRAGMENTBUNDLESERVICE	Traces bundle cache refresh, storing in cache, bundle not found, base bundle used, fragment tag rendered

Tracing components and applications

Table 96, page 484 describes the flags in com.documentum.web.form.Trace that can be used to components and applications.

Table 96. Component and application tracing flags

Flag name	Description
Components	COMPONENT traces component launching, role precondition check, nesting, form class and NLS bundle, and dispatching context, start page.
RESOURCE_CLASS	Traces missing strings from NLS resource bundle when an NLS key is present in the JSP page but no value for the key is present in the NLS properties file. Output has the form 'yyMSG_TITLEyy'.
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

Flag name	Description
VDMTREEGRID	Traces temporary annotation file creation, path, and errors in retrieving and alternative URL; failure to extract format, name of file streamed, end of stream and byte size; imaging servlet operations
SEARCH	Traces failure to open saved search results; search results columns; last search ID
CLUSTERING	Traces whether query is not finished yet; query status has changed; new query results are received; query has completed
COLLABORATIONSERVICE	Traces license check for a BOF 2 global registry (which is set in dfc.properties)

Tracing virtual links

Table 97, page 485 describes the flags in com.documentum.web.virtuallink.Trace that can be used to trace virtual linking.

Table 97. Virtual link tracing flags

Flag name	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.
VIRTUALLINKMATCHING	Traces virtual link path and path exception, partial match path.
VIRTUALLINKREQUEST	Traces request string and writes details to the log.

Tracing servlets

Table 98, page 486 describes the flags in `com.documentum.web.servlet.Trace` can be used to trace servlet operations.

Table 98. Servlet tracing flags

Flag name	Description
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

Table 99, page 486 describes the tracing flags in `com.documentum.job.async.Trace` and `com.documentum.job.Trace` that can be used to trace asynchronous jobs.

Table 99. Asynchronous operation tracing flags

Flag name	Description
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

Table 100, page 487 describes the flags in `com.documentum.web.common.Trace`, `com.documentum.web.contentxfer.Trace`, and `web.util.Trace` that can be used to trace content transfer and ACS transfer operations.

Table 100. Content transfer tracing flags

Flag name	Description
ZIPARCHIVE	Traces the zip archive that is created to transfer content including stream open, close, filename, and folder name.
GETCONTENT	Traces servlet initiation and completion, object retrieval, format test, extraction, streaming, and writing to local host.
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
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
EMFIMPORT	Traces the importing of email messages as the <code>dm_message_archive</code> type.
TRUSTEDDOMAIN	Traces the validation of the host of a given URL (which is usually used in redirects).

WDK trace flags to trace Test Harness at runtime

Two WDK trace flags are defined to trace the Test Harness at runtime:

- `com.documentum.web.test.Trace.TESTCASE`: This flag traces the traversal of test suites and test cases and the execution of test cases.
- `com.documentum.web.test.Trace.TESTSTEP`: This flag traces the traversal of test steps and the execution of test steps.
- `com.documentum.web.test.Trace.TESTRECORDER`: This flag traces recorder preferences, listeners, and other recorder actions.

Configuration Settings in WDK-based Application Deployment

The following tables list the mandatory and optional configuration elements that can be set before, during, and after deployment of WDK-based applications.

[Table 101, page 489](#) lists the configuration elements that must be set before deploying a WDK-based application such as Webtop or TaskSpace. Not all of these elements must be set for every deployment, but if you wish to support the function in the first column, you must enable it before deployment. Refer to *Web Development Kit and Webtop Deployment Guide* for more information on these settings.

Because WDK-based applications encapsulate DFC, you can also configure DFC settings as described in `dfcfull.properties`. This file is located in the `WEB-INF/classes` directory of the WDK-based application.

Functions marked with an asterisk (*) must be performed for every deployment.

Table 101. Mandatory configuration before deployment

Function	Element	Location
Turn off tag pooling (Tomcat, Oracle)*	<code>servlet.init-param</code>	<code>web.xml</code>
Global registry indications*	<code>dfc.docbroker.host</code> <code>dfc.globalregistry.repository</code> <code>dfc.globalregistry.username</code> <code>dfc.globalregistry.password</code>	<code>dfc.properties</code> in <code>WEB-INF/classes</code>
WAS compiler and classloader*	Classloader order <code>useJDKCompiler</code>	WAS admin console

[Table 102, page 490](#) lists the configuration settings that can be set before deployment.

Table 102. Optional configuration before deployment

Function	Element	Location
WAS failover	NoAffinitySwitchBack	WAS cluster configuration
WAS global security	Security policies and environment variables	Download from Powerlink if needed
UCF to use file, not Windows registry	registry.mode	ucf.installer.config.xml
Default content transfer directories for client	option	ucf.installer.config.xml
Unsigned SSL certificates	option	ucf.installer.config.xml
Proxy servers	http11.chunked.transfer	ucf.server.config.xml in WEB-INF/classes
Set content transfer mode	contentxfer.default-mechanism	custom/app.xml
Change ACS and BOCS behavior	contentxfer.accelerated-read contentxfer.accelerated-write	custom/app.xml
Locale settings	language	custom/app.xml
Java EE principal authentication	principal credentials securityconstraint	TrustedAuthenticator-Credentials.properties in WEB-INF/classes/com/documentum/web/formext/sessionweb.xml in WEB-INF

Table 103, page 490 lists the configuration settings that can be changed after deployment.

Table 103. Optional configuration after deployment

Function	Element	Location
Turn off failover	failover.enabled	custom/app.xml
Change presets repository (default = global registry)	presets	custom/app.xml
Change preferences repository (default = global registry)	preferencesrepository	custom/app.xml
Default repository	authentication.docbase	custom/app.xml
Allow saved credentials	save_credentials	custom/app.xml
Encrypt passwords for drl, drlauthenticate, and virtuallinkconnect		com.documentum.web.formext.session.TrustedAuthenticatorTool

Function	Element	Location
Enable accessibility	accessibility	custom/app.xml
Configure app server timeout	sessionconfig	web.xml in WEB-INF

508, *see* accessibility

A

Accelerated Content Services

configuration, 48

accessibility

buttons and icons, 455

configuration in app.xml, 70

control label, 453

development guidelines, 453

frames, 454

images, 453

login, 455

service, 453

<accessibility>, 70

ACS, 48

tracing, 487

action

remove from component, 221

actions

configuration file, 219

custom, 232

customizing, 227

dynamic state, 122

execution, 239

generic, 122, 231

genericnoselect, 122

in right-click menu, 134

introduction, 216

LaunchComponent, 231

LaunchComponent navigation, 256

launching, 216

launching components, 255

listeners, 241

multiselect, 122

nesting, 245

overview, 215

passing arguments, 218

pre- and post-processing, 245

preconditions, 236

role-based, 423

roles and object ACL permissions, 424

singleselect, 122

startup, in URL, 258

tracing, 216, 481

version, 250

<adobe_comment_connector>, 103

alternative.chunking.buffer.size, 63

alttextenabled, 70

<always-show-type-icon>, 109

AppFolderName, 110

Application Connectors

authentication, 278

components, 274

configuring, 269

events, 277

application layer

inheritance, 114

overview, 116

application server

slows down, 457

application stops working, 457

ApplicationLocation, 477

qualifier, 44

applications

environments, 104

failover, 86

inheritance, 114

interaction of elements, 429

login, 460

managing frames, 176

name, 110

performance, 437

properties, 104

qualifier, 43

timeout, 74

tracing, 484

architecture, Documentum stack, 25

ArgumentTag, 283

ArrayResultSet, 209

asynchronous

- actions, 364
- components, 366
- framework, 367
- global settings, 91
- listeners, 367
- overview, 91
- process, 369
- UI, 368
- <at least one index entry>, 212
- attributelist
 - configuration file, 198
 - context or scope, 197
 - controls, 196
- attributes
 - custom data handler, 210
 - custom data handlers, 96
 - display of, 162
 - in search, 387
 - list in data dictionary, 197
 - required, 162
 - single and repeating, 195
 - tracing, 482
- authentication
 - elements, in app.xml, 64
 - Java EE, 67
 - overview, 369
 - schemes, 381
- <authentication>, 64
- autocompletion
 - adding support to controls, 161
 - configuring, 134
 - enabling, 81
 - tracing, 482

B

- binding
 - data, 186
 - to a thread, 375
- BOF, in WDK, 318
- bookmarks, 471
- branding
 - images and icons, 360
 - overview, 356
 - style sheets, 358
 - tracing, 481
- breadcrumb
 - tracing, 482
- browser
 - requirements, 105

- <browserrequirements>, 105
- browsertree
 - custom, 472
 - limiting size, 104
- business objects, in WDK, 318
- buttons, 362
 - accessible, 455
 - changing text, 362

C

- cache size
 - query, 439
- caching
 - data, 207
 - folder names, 374
 - formats, 374
- case sensitivity
 - in WDK basic search, 390
- certificates, 59
- client capability
 - overview, 423
 - roles, 72
- client session state
 - global setting, 108
- <client_capability_aliases>, 74
- <client-sessionstate>, 108
- ClientCacheControl, 112
- clientenv
 - in scope, 45
 - qualifier, 43
- clipboard
 - overview, 297
 - tracing, 481
 - using in a component, 297
- clusters
 - failover, 86
- collaboration service
 - tracing, 485
- columns
 - configuring, 260
 - dynamic (runtime), 262
 - number of, 193
- combocontainer, 313
- comment
 - PDF Annotation Services,
 - enabling, 103
- component
 - remove action, 221
- componentinclude, 258

- components
 - APIs, 280
 - calling components from URL, 304
 - calling from action, 305
 - calling from another component, 257
 - Component class, 323
 - containers, 304
 - definition, 321
 - filters, 253
 - getting reference in JSP, 302
 - hiding with notdefined, 252
 - hiding with scope, 252
 - implementation, 285
 - implementing failover, 292
 - included, 258, 285
 - inheritance, 249
 - invoking within a container, 304
 - jump to, 281
 - launch by action, 255
 - lifecycle, 324
 - listener, 295
 - messages and labels, 265
 - overview, 249, 319
 - role-based UI, 426
 - scope, 251
 - tracing, 484
 - UCF, configuring, 413
 - UI, 323
 - URL to, 254
 - using the clipboard, 297
 - validation, 324
 - version, 250
- compression.exclusion.formats, 62
- CompressionFilter, 112
- ConfigResultSet, 208
- ConfigService, 431
- configuration
 - classes, 430
 - component XML file, 321
 - control tags, 126
 - files, finding, 124
 - inheritance, 40
 - lookup algorithm, 437
 - object filter example, 139
 - processing, 436
 - scope (qualifiers), 44
 - service, 430
 - tabs, 129
 - tag libraries, 126
 - tracing, 481
 - WDK, 489
- configuration file
 - action, 219
 - attributelist, 198
 - component, 321
 - deleting elements, 37
 - inserting into, 35
 - lookup, 432
 - modification syntax, 33
 - modifying extended definitions, 37
 - replacing elements, 35
- containers
 - accessing contained components, 311
 - calling components in a container, 304
 - calling contained components from
 - script, 305
 - calling from server class, 306
 - change notifications, 307
 - classes, 317
 - components that require, 317
 - modal, 315
 - navigate to next page, 312
 - navigating, 309
 - overview, 304
 - propagating values within, 309
 - tracing, 484
- content transfer
 - classes, 419
 - debugging, 420
 - listeners, 418
 - modes, 47
 - modes, compared, 407
 - progress, 420
 - registry, 61
 - results (UCF), 418
 - service and processor classes, 414
 - stream to browser, 412
 - temporary, on application server, 62
- ContentTransferService, 414
- <contentxfer>, 48
- context
 - attributelist, 197
 - overview, 435
 - passing values to action, 219
 - relation to scope, 44
- context menus, 134
 - enabling, 81
- contextvalue attribute, 219
- Control class
 - base class, 146

- control tag
 - configuration, 126
 - tracing, 482
- controls
 - accessing, 153
 - action-enabled, 121
 - attribute lists, 196
 - boolean, 120
 - changing a style, 360
 - choosing a superclass, 153
 - component helper, 121
 - configuration overview, 126
 - contained, looping through, 155
 - Control base class, 146
 - ControlTag class, 147
 - databound, 186
 - databound, see databound
 - controls, 186
 - events, 172
 - format, 121
 - handling events on client, 177
 - handling events on server, 173
 - hiding, 146
 - ID, 153
 - indexed, 154
 - media, 121
 - multiple selection, 159
 - naming, 154
 - overview, 119
 - passing arguments to event
 - handler, 283
 - relation to tags, 149
 - retrieving values, 153
 - scrollable, 142
 - search, 389
 - string input, 120
 - tooltips, 144
 - tracing, 481
 - types, 120
 - validation, 133
 - XML configuration file, 127
- ControlTag, 147
- cookies
 - writing and reading, 334
- copy_operation
 - in app.xml, 93
- <copy_operation>, 93
- cross-site scripting
 - HTML in attribute values, 349
 - validation, 106
- CSV
 - adding support for export, 291
- <custom_attribute_data_handlers>, 96
- customization
 - validator, 157
 - Webtop browsertree, 472
- D**
- data dictionary
 - attribute labels, 346
 - list of attributes, 196
 - scopes, displayed in WDK, 197
 - using attribute lists, 197
- data drop-down lists
 - configuring, 194
- databound controls
 - binding data, 186
 - caching data, 207
 - DataProvider class, 204
 - getting data, 204
 - introduction, 186
 - paging, 193
 - result sets, 208
 - ResultSet data provider, 204
 - sorting, 192
- datagrid
 - adding custom attributes, 210
 - column resizing, 186
 - fixed column headers, 186
- <datagrid>, 81
- datagridRow
 - configuring row selection, 188
- datagridRowModifier, 190
- DataProvider
 - class, 204
 - getXXX, 207
 - setQuery, 204
- debug_preferences, 331
- debugging
 - content transfer, 420
 - preferences UI, 331
- deep export
 - element, *see italicstest*
- defaultonenter event, 151
- deployment descriptor
 - listeners, 114
 - servlets, 112
- <desktopui>, 81
- DFC

- storing objects, 374
- directory structure
 - applications, 116
 - branding, 356
 - Webtop, 465
- <discussion>, 90
- display
 - global settings, 90, 92
- <display>, 90, 92
- docbase
 - qualifier, 42
- DocbaseAttributeCache, 375
- docbaseattributelist control, 196
- <docbaseicon>, 109
- docbaseobject
 - configuration, 162
- DocbaseObjectCache, 375
- docbaseobjectconfiguration
 - definition, 162
- <docbaseobjectconfiguration>, 167
- drag and drop
 - global setting, 98
 - in component definition, 298
 - in control, 301
 - in JSP, 299
 - overview, 298
 - performance, 299
 - troubleshooting, 301
- <dragdrop>, 98, 298
- dragdropregion tag, 299
- drop-down lists
 - configuring, 194
 - customizing, 160
- dynamicAction.js, 122

E

- email
 - enable EMCMF format in WDK, 78
 - importing, 77
- <email-format>, 77
- entitlement
 - qualifier, 44
- Environment.properties, 104
- environments
 - introduction, 104
- error message
 - configuration in app.xml, 96
 - service, 289
- <errormessageservice>, 96

- event
 - Content Server, notifications, 85
- event handler
 - navigation, 178
 - navigation to component, 178
 - registering client handlers, 179
- events
 - between frames, 181
 - client preprocessing, 265
 - client to server, 174
 - client-side, 172
 - control, 172
 - handle on server or client, 152
 - handling on client, 177
 - handling on server, 173
 - how they are raised, 151
 - server to client, 176
- execution
 - class, 239
 - element, 220
 - permit, 221
- <execution>, 220
- export
 - to CSV, customizing, 291
- extended search, 389

F

- failover, 86
 - enabling for the application, 86
 - implement in component, 292
 - tracing, 480
- <failover>, 74
- <fallback_identity>, 74
- favorites, 471
- file.poll.interval, 62
- filters, 253
 - based on role, 425
 - dynamic (with
 - LaunchComponent), 233
 - latency/bandwidth, 443
 - servlet, 111
 - starts with, 201
- fixed headers
 - columns, 186
- flags, tracing, 479
- folders
 - caching, 374
- foreign objects
 - description, 380

- operations on, 265
- foreign type, 222
- form processor
 - properties, 94
- formats
 - caching, 374
- <formats>, 100
- forms
 - form processor, 324
 - tag class, 324
- FormTag, 324
- fragment control, 143
- <fragmentbundles>, 143
- frames
 - accessibility, 454
 - firing events, 181
 - managing, 176
- full-text search
 - in WDK, 388

G

- generic, 122
- generic actions, 231
- genericnoselect, 122
- <groupclasses>, 71

H

- help service
 - calling, 268
- <hiddenobject>, 90
- <hideinvalidactions>, 90
- high availability
 - configuring in application, 74
- history
 - configuring, 95
 - performance, 442
- hooks
 - form navigation, 94
 - lookup, 433
- hotkeys
 - configuring, 222
 - customizing, 226
 - customizing controls, 226
 - enabling, 80
 - key combination map, 224
 - tracing, 482
 - XML definition, 223
- <hotkeys>, 223

- HotKeysNlsProp, 224
- HTML
 - in attributes, 349
 - rendering, 170
- HTTP
 - compared to UCF, 407
- HttpContentTransportManager, 410
- https
 - certificates, 59

I

- IApplicationListener, 376
- IAuthenticationScheme, 381
- IConfigElement, 431
- IConfigLookup, 432
- IConfigLookupHook, 433
- IConfigReader, 434
- icons
 - controls, 145
 - for custom type, 360
 - themes, 360
- IDfSession, 371
- IDfSessionManager, 372
- IDfSessionManagerEventListener, 376
- ILaunchComponentEvaluator, 233
- images, 145
 - buttons, 362
 - in style sheets, 361
 - replacing, 145
 - themes, 360
- imaging
 - tracing, 485
- inbox
 - component group, 468
- included component, 285
- <infomessageservice>, 96
- inheritance
 - applications, 114
 - components, 249
 - configuration, 40
 - strings, 347
- <init-controls>, 412
- initialize
 - content transfer controls, 412
- interaction
 - of application elements, 429
- internationalization
 - testing, 350
- introduction

- WDK Automated Test Framework, 23
 - IQualifier, 46
 - IRequestListener, 376
 - IReturnListener, 295
 - ISessionListener, 376
- J**
- Java EE authentication, 67
 - JavaScript, 457
 - files in WDK, 180
 - modal windows, 303
 - postServerEvent, 174
 - registering, 180
 - tracing, 181
 - using, 179
 - utilities, 172
 - <job-execution>, 91
 - JSP
 - creating, 264
 - fragments, 143
 - getting component reference, 302
 - jump to page, 280
 - request tracing, 480
 - standard, 24
 - structure of page, 323
 - using outside components, 264
 - jump
 - to component, 281
 - to component page, 280
- K**
- keyboardnavigationenabled, 70
 - keystore, 59
- L**
- language
 - element, in app.xml, 65 to 66
 - <language>, 65 to 66
 - latency
 - performance filters, 443
 - LaunchComponent, 231, 255, 305
 - dynamic filters, 233
 - lifecycle
 - component, 324
 - lightweight sysobject
 - configuring, 97
 - <lightweight-sysobject>, 97
 - links
 - enabling with row selection, 190
 - hiding with row selection, 190
 - rendering, 171
 - listeners
 - action, 241
 - application, 375
 - component, 295
 - content transfer, 418
 - control, 169
 - in web.xml, 114
 - request, 375
 - session, 375
 - session manager, 376
 - <listeners>, 96
 - ListResultSet, 209
 - lists, drop-down
 - configuring, 194
 - customizing, 160
 - load balancing, 445
 - <locale>, 66
 - locales
 - adding, 347
 - in app.xml, 66
 - login, 67
 - NLS file naming, 348
 - NLS files, 347
 - retrieving strings, 349, 354
 - tracing, 480
 - location
 - qualifier, 44
 - logging, 448
 - UCF, 420
 - login, 460
 - accessible, 455
 - Java EE principal, 67
 - locale, 67
 - password encryption, 69
 - setting up Java EE principals, 68
 - silent, 383
 - skip authentication, 462
 - ticketed, 460
- M**
- memory allocation, 440
 - <menu>, 135
 - menus
 - context, 134
 - fixed, 135
 - passing arguments to, 229

- message service, 287
- mirror object, 380
- modal pop-up windows
 - enabling, 77
- modal windows
 - overview, 303
 - performance, 446
- modification
 - deleting, 37
 - inserting, 35
 - replacing, 35
 - syntax, 33
- <modified_vdm_nodes>, 75
- modifies
 - in config file, 33
- move_operation
 - in app.xml, 93
- <move_operation>, 93
- multiple selection
 - actions, 122
 - implementing, 159

N

- navigation, 458
 - in client event handler, 178
 - LaunchComponent, 256
 - return to caller, 282
 - to a component, 178
 - to next page, 312
 - URL parameters, 151
 - URL to component, 254
 - Webtop, 476
 - within containers, 309
- network disks
 - optimizing transfer, 56
- NLS
 - definition, 346
 - dynamic substitution, 355
 - filenames, 348
 - includes, 347
 - strings, 347
 - tracing missing strings, 484
- notdefined, 252
- <notification>, 85
- notifications
 - Content Server event, 85

O

- objectfilter
 - example, 139
- olecompound element, 221
- onchange event, 150
- onclick event, 151
- onselect event, 150

P

- page not found
 - HTTP 1.1 not enabled, 458
- paging
 - of data, 193
- password encryption, 69
- PDF Annotation Services
 - enabling, 103
- performance
 - actions, 438
 - browser history, 442
 - cookies, 442
 - drag and drop, 299
 - events, 438
 - HTTP sessions, 441
 - import, 445
 - latency/bandwidth filters, 443
 - memory settings, 440
 - modal windows, 446
 - object creation, 439
 - overview, 437
 - paging and cache size, 439
 - preferences, 442
 - qualifiers, 445
 - queries, 438
 - strings, 439
 - suppress folder path display, 394
 - suppress summary calculation, 395
 - tracing, 438
 - value assistance, 442
- phishing
 - security, 84
- plugins
 - in app.xml, 97
- <plugins>, 97
- pop-up, 313
- postComponentJumpEvent(), 178
- postComponentNestEvent(), 178
- postServerEvent(), 174
- precondition
 - element, 220

- overview, 236
- passing argument to, 228
- role, 220
- <precondition>, 220
- preferences
 - and cookies, 329
 - configuration, 337
 - definition, 337
 - repository, configuring, 83
 - storing and retrieving, 334
 - tracing, 481
 - user, configuration, 330
 - user, configuring, 331
- <preferences>, 337
- PreferenceStore, 334
- <preferred_renditions>, 101
- presets
 - enabling, 82
 - in app.xml, 82
 - in custom components, 344
 - overview, 343
- primary element, 32
- primary folder path
 - displaying, 172
 - getting, 171
- principal (Java EE) authentication, 67
- privilege
 - qualifier, 43
- progress
 - content transfer, 420
- prompt, 313
- properties
 - not updated, 456
- properties files
 - accessibility support, 453
 - configuring, 347
 - dynamic substitution in, 355
 - inheritance, 347
 - naming, 348
 - overriding strings, 349
 - overview, 346
 - retrieving strings, 354
 - string externalization, 346
- proxy
 - reverse, configuration for, 63
- proxy client access, 51
- <pseudo_attributes>, 140
- pseudoattributes, 140

Q

- qualifiers
 - clientenv, 45
 - element in app.xml, 96
 - performance, 445
 - resolving to scope, 46
 - Webtop, 467
- <qualifiers>, 96
- queryExecute(), 236

R

- <readnotification>, 85
- redirect
 - security, 84
- reference object, 380
- refresh
 - data, 207
 - value assistance, 124
 - XML, 31
- registerClientEventHandler(), 179
- registry
 - in content transfer, 61
- rendering
 - control value, 170
 - HTML, 170
 - links, 171
- repeating attributes
 - editing, 195
- replica object, 380
- repository
 - qualifier, 42
- RequestAdapter, 112
- <requestvalidation>, 106
- required attribute
 - configuring, 162
- requiresVisit, 306
- resizing
 - columns, 186
- result sets, 208
 - ArrayResultSet, 209
 - ConfigResultSet, 208
 - ListResultSet, 209
 - ScrollableResultSet, 208
 - TableResultSet, 209
- return navigation, 282
- return values
 - from a container, 296
- <reverse-precondition>, 221
- rich text

- configuration, 140
 - richtexteditor
 - in app.xml, 99
 - <richtexteditor>, 99
 - right to left
 - enable, 66
 - <rolemodel>, 71
 - roles
 - actions based on, 423
 - client capability plugin, 72
 - component filter, 425
 - configuration, 427
 - constraints, 424
 - in Webtop, 423
 - model in app.xml, 71
 - precedence, 424
 - qualifier, 43
 - role model adaptor, 426
 - tracing, 481
 - UI based on, 426
 - <rolesprecedence>, 424
 - row selection
 - configuring, 188
 - enabling, 81
 - <rowselection>, 81
 - runatclient attribute, 150
- S**
- SafeHTMLString, 449
 - saved searches
 - in WDK, 387
 - scope
 - and qualifiers, 44, 46
 - attributelist, 197
 - components, 251
 - foreign, 222
 - hiding components, 252
 - hiding subtypes, 222
 - scrollable controls
 - configuring, 142
 - ScrollableResultSet, 208
 - search
 - advanced, configuring, 391
 - attribute values, 387
 - basic, configuring in WDK, 390
 - configuring controls, 389
 - customizations in Webtop, 396
 - full-text, in WDK, 388
 - in WDK, overview, 387
 - number of results, 394
 - results display, 395
 - results, configuring in WDK, 394
 - saved searches, 387
 - sources in WDK, 387
 - term hit highlighting, 394
 - tracing, 485
 - tracing clusters, 485
 - user preferences, configuring, 395
 - value assistance, 388
 - security
 - preventing redirects, 84
 - trusted domains, 65
 - server.session.timeout.seconds, 62
 - servlets
 - filters, 111
 - in web.xml, 112
 - mapping in web.xml, 109
 - standard, 24
 - WDKController, 292
 - <session_config>, 75
 - SessionManagerHttpBinding, 372
 - sessions
 - configuration in app.xml, 75
 - cookies, 445
 - HTTP and repository, 369
 - IDfSessionManager, 372
 - repository, 371
 - running out of, 458
 - SessionManagerHttpBinding, 372
 - state, 84, 374
 - tracing, 381
 - setComponentJump(), 306
 - setComponentNested(), 306
 - setComponentReturn, 282
 - setReturnError, 289
 - shortcutnavigationenabled, 70
 - shortcuts
 - enabling, 80
 - key combination map, 224
 - Show All Properties, 456
 - showifinvalid, override, 90
 - singleselect, 122
 - skip authentication, 462
 - SSL
 - configuring UCF support, 59
 - "starts with" filter, 201
 - startupAction, 216, 258
 - StaticPageExcludes, 86
 - StaticPageIncludes, 86

- storing objects, 374
 - streamline view
 - enabling, 85, 465
 - <streamlineviewvisible>, 465
 - strings
 - configuration overview, 346
 - dynamic substitution in, 355
 - inheritance, 347
 - retrieving, 349, 354
 - StringUtil, 450
 - style sheets
 - changing a control style, 360
 - images, 361
 - introduction, 358
 - modifying, 359
 - order of precedence, 358
 - <supported_locales>, 66
- T**
- tab key order, 130
 - TableResultSet, 209
 - tabs
 - configuration example, 129
 - tags
 - attributes, 148
 - base classes, 147
 - generating UI, 170
 - relation to controls, 149
 - using libraries, 126
 - task manager
 - component group, 468
 - temp.working.dir, 62
 - term hit highlighting
 - in WDK search, 394
 - test harness
 - trace flags, 488
 - testing
 - internationalization, 350
 - themes
 - configuration in app.xml, 70
 - definition, 357
 - directory structure, 356
 - images and icons, 360
 - overview, 356
 - processing, 363
 - <themes>, 70
 - thread
 - binding and caching, 375
 - tracing, 479
 - ThreadLocalCache, 375
 - ThreadLocalVariable, 375
 - timeout
 - control, 75
 - for long events, 75
 - Java EE server setting, 74
 - overriding, 75
 - troubleshooting, 76
 - virtual document checkin, 75
 - toolbar
 - re-enabling, 467
 - tooltips, 144
 - tracing, 216
 - ACS, 487
 - adding flags, 447
 - asynchronous jobs, 486
 - client-side, 447
 - clipboard, 481
 - components and applications, 484
 - content transfer, 487
 - controls, 481
 - dfc, 447
 - framework operations, 480
 - JavaScript, 181
 - JSP processing, 482
 - overview, 446
 - performance, 438
 - sessions, 381, 479
 - test harness, 488
 - turning on, 446
 - UCF, 487
 - validation, 481
 - virtual links, 485
 - WDK flags, 479
 - tracing.enabled, 62 to 63
 - troubleshooting
 - Application Connectors
 - connection, 279
 - roles, 424
 - <trusted-domains>, 65
 - truststore, 59
 - type
 - qualifier, 42
- U**
- UCF
 - client config files, 52
 - client configuration, 53
 - client path substitution, 58

- client services, 52
- compared to HTTP, 407
- components, configuring, 413
- customization, 414
- logging, 420
- overview, 412
- results, 418
- server config files, 62
- tracing, 487
- troubleshooting, 415
- <ucfrequired>, 62
- UcfSessionInit, 112
- URL
 - in JSP pages, 323
 - to component, 254
 - to component in container, 255
 - virtual link, 90
- user.home, 53

V

- validation, 133
 - base control class, 157
 - by form processor, 156
 - custom control, 157
 - process, 156
 - tracing, 482
- value
 - rendering, 170
- value assistance
 - in WDK search, 388
 - non-data dictionary, 213
 - overview, 123
 - performance, 442
 - refresh, 124
- version
 - components and actions, 250
 - qualifier, 43
- view
 - content in browser, 412
- views
 - classic, 466
 - streamline, 467

- Webtop, 465
- virtual document
 - tracing the tree, 484
- virtual links
 - authentication, 87
 - overview, 89
 - path resolution, 89
 - URL, 90

W

- WDK
 - enable EMCMF format, 78
- WDKController, 112, 292
- web.xml
 - filters, 111
- WebformTag, 324
- Webtop
 - bookmark URLs, 471
 - classic view, 466
 - configuration overview, 465
 - custom browsertree node, 472
 - customizing, 465
 - directory structure, 465
 - navigation, 476
 - qualifiers, 467
 - redirecting to, 472
 - streamline view, 467
 - views, 465
- WebtopComponentUtil, 477

X

- <xforms>, 96
- XML
 - file extensions, configuring, 101
 - file import, 458
- <xmlfile_extensions>, 101
- XSS, *see* cross-site scripting

Z

- ZipArchive, 451