Novell exteNd Director

5.0

www.novell.com

CONTENT MANAGEMENT GUIDE

Novell®

Legal Notices

Novell, Inc.

1800 South Novell Place

Provo, UT 85606

www.novell.com

exteNd Director *Content Management Guide*

December 2003

Online Documentation: To access the online documemntation for this and other Novell products, and to get updates, see www.novell.com/documentation.

Novell Trademarks

ConsoleOne is a registered trademark of Novell, Inc.
eDirectory is a trademark of Novell, Inc.
GroupWise is a registered trademark of Novell, Inc.
exteNd is a trademark of Novell, Inc.
exteNd Composer is a trademark of Novell, Inc.
exteNd Director is a trademark of Novell, Inc.
iChain is a registered trademark of Novell, Inc.
jBroker is a trademark of Novell, Inc.
NetWare is a registered trademark of Novell, Inc.
Novell is a registered trademark of Novell, Inc.
Novell eGuide is a trademark of Novell, Inc.

SilverStream Trademarks

SilverStream is a registered trademark of SilverStream Software, LLC.

Third-Party Trademarks

Acrobat, Adaptive Server, Adobe, AIX, Autonomy, BEA, Cloudscape, DRE, Dreamweaver, EJB, HP-UX, IBM, Informix, iPlanet, JASS, Java, JavaBeans, JavaMail, JavaServer Pages, JDBC, JNDI, JSP, J2EE, Linux, Macromedia, Microsoft, MySQL, Navigator, Netscape, Netscape Certificate Server, Netscape Directory Server, Oracle, PowerPoint, RSA, RSS, SPARC, SQL, SQL Server, Sun, Sybase, Symantec, UNIX, VeriSign, Windows, Windows NT

All third-party trademarks are the property of their respective owners.

Third-Party Software Legal Notices

The Apache Software License, Version 1.1

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)."

Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.

4. The names "Apache" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Autonomy

Castor

Redistribution and use of this software and associated documentation ("Software"), with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain copyright statements and notices. Redistributions must also contain a copy of this document.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name "ExoLab" must not be used to endorse or promote products derived from this Software without prior written permission of Intalio Inc. For written permission, please contact info@exolab.org.

4. Products derived from this Software may not be called "Castor" nor may "Castor" appear in their names without prior written permission of Intalio Inc. Exolab, Castor and Intalio are trademarks of Intalio Inc.

5. Due credit should be given to the ExoLab Project (http://www.exolab.org/).

THIS SOFTWARE IS PROVIDED BY INTALIO AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL INTALIO OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Indiana University Extreme! Lab Software License

Version 1.1.1

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Indiana University Extreme! Lab (http://www.extreme.indiana.edu/)."

Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.

4. The names "Indiana University" and "Indiana University Extreme! Lab" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact http://www.extreme.indiana.edu/.

5. Products derived from this software may not use "Indiana University" name nor may "Indiana University" appear in their name, without prior written permission of the Indiana University.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS, COPYRIGHT HOLDERS OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

JDOM.JAR

Phaos

Sun

W3C

3. Notice of any changes or modifications to the files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permission. Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

Contents

About This Guide **15**

PART I CONCEPTS **17**

1 About the Content Management Subsystem **19**
About content management 19
About content 20
About documents 20
Content and pages 21
Subsystem infrastructure 21
Physical infrastructure 22
Logical infrastructure 23
Defining content structure and layout 24
Classifying content 25
Content life cycle 26
Checking out documents 26
Publishing a document 26
Subsystem support functions 26
Integration with other subsystems 27

2 Developing Content Management Infrastructure **29**
About the CM API 30
Getting a content manager object 30
Changing repository data 30
About the CM subsystem infrastructure 31
Managing fields 32
Adding a field 34
Adding a field to a portlet 34
Listing fields using different filters 35
Managing document types 35
Adding a document type with associated fields 37
Managing layout styles 38
User agents 41
Adding a layout style 41
Adding a layout document and a layout document descriptor 42
Changing a layout style 43
Managing folders and categories 43
Adding a category 45
Navigating the CM hierarchy 46

3 Managing Documents 49
About documents 49
Accessing the CM API 50
Adding documents 50
Adding a document 51
Adding multiple documents 52
Specifying field values for a document 54
Getting fields for the document type 54
Getting a field object by name 55
Setting a field value 55
Getting all fields 57
Getting field values for a single field 57
Specifying layout sets for documents 60
When to use a layout set 60
Methods for managing layout sets 61
Creating links between documents 62
Two types of document relationships 62
Hierarchical linking 63
Adding a child document 64
Compound linking 65
Linking a child document 67
Updating a link with a new document version 67
Getting linked parent documents 68
Getting linked child documents 68
Modifying and publishing documents 69
Tracking document status 70
Methods for source control and publishing 71
Displaying documents 72
HTML content 72
XML content 73
Composite documents 74

4 Securing Content 77
About access control 77
CM user groups 78
ACL-based security 78
Permissions 79
Element types and associated permissions 79
ContentAdmin group 80
Methods for managing access control 80
Accessing ACLs for existing elements 80
Specifying ACLs for new elements 81
Inheriting ACLs 81
Accessing ACLs for ContentAdmin 82
Restricting element access to administrators 82
Examples of adding ACLs 82
Example of handling a security exception 84

5 Managing Tasks 85
About tasks 85
Installed tasks 86
Custom tasks 86
About how tasks are registered and configured 87
tasktypes.xml 87
Default_tasklist.xml 88
services.xml 89
Customizing an installed task 89
Creating and implementing a new task 90
Custom task sample code 93
NewDocumentNotifier 93
PeriodicNewDocumentNotifier 100
Working with task events 102
Task event types 102
Registering for a task event 103
Enabling or disabling a task event 103

6 Managing Content Caching 105
About caching in CM 105
Summary of CM caching information 106
Caching behavior 106
Caching of folders, categories, and document metadata 107
About document content and versions 107
Controlling caching in the DAC 107

7 Importing and Exporting Content 109
About importing and exporting 109
Using the import/export facilities 110
About the export facility 110
Export process 111
About the import facility 112
Import process 112
Customizing imports and exports 113
Customizing the data export descriptor (DED) 113
Customizing the data import descriptor (DID) 114
Accessing the import and export API 114

8 Working with Content Management Events 115
About CM events 115
CM event types 115
Registering for CM events 119
Registering for events on directory elements 119
Specifying event types 120
Using the event helper class 121
Event registration examples 121
Enabling CM events 123

PART II WEBDAV 125

9 Using WebDAV Clients with exteNd Director for Collaborative Authoring 127
What is WebDAV? 127
Information elements for distributed Web authoring 128
WebDAV extensions to HTTP 128
About exteNd Director's WebDAV support 129
What you can do with the exteNd Director WebDAV subsystem 130
How exteNd Director stores content from WebDAV clients 130
How exteNd Director secures content from WebDAV clients 131
How exteNd Director manages versioning for WebDAV clients 131
Installing the exteNd Director WebDAV subsystem 132
Deploying the exteNd Director WebDAV subsystem 133
Before you deploy 133
Setting up the client 133
Supported WebDAV methods 134
Public WebDAV server 135

10 Building Your Own WebDAV Client 137
About the WebDAV client API 137
Why build your own WebDAV client? 138
Configuring your environment 138
Using the WebDAV client API 139
WebDAV requests and responses 140
Working with resources, collections, and properties 140
Classes 141
Helper methods 141
Utility methods 142
Programming practices 144
Programming practices using helper methods 144
Programming practices using utility methods 146
Issuing WebDAV requests from a Java client 149
Adding a category reference to a document 150
Copying a resource or collection 154
Creating a new collection 155
Creating a new document from a custom template 156
Deleting a document 158
Getting a resource or collection 158
Getting header information from a resource or collection 160
Getting methods that can be called on a resource or collection 162
Getting properties defined on a resource or collection 164
Locking a document 166
Moving a resource or collection 168
Removing a category reference from a document 169
Removing all category references from a document 172
Renaming a resource or collection 175
Setting the value of a custom field in a document 177

Unlocking a document 180
Updating a document 181

11 Working with WebDAV Events 183
About WebDAV events 183
Event types 183
Registering for WebDAV events 184
Enabling WebDAV events 185

PART III CMS ADMINISTRATION CONSOLE 187

12 About the CMS Administration Console 189
What CM tasks you can do with the CMS Administration Console 189
How to access the CMS Administration Console 192
The main CMS Administration Console page 193
Interactive controls 194

13 Setting Up the Required Infrastructure 197
Flow of operations 197
Creating folders 198
Creating document types 199
Creating fields and adding them to a document type 202
About fields 202
Creating and manipulating fields 203
Writing JavaScript for document types and fields 206

14 Setting Up the Optional Infrastructure 211
Flow of operations 211
Creating display styles 212
About display styles 212
Specifying a style sheet for a document type 217
Creating taxonomies 218
Creating categories 219

15 Creating Content 221
About content 221
Flow of operations 222
Creating documents 223
Creating a document 224
Specifying a folder for a new document 228
Using Auto Create to create a document 229
Using the CMS Administration Console's HTML Editor 229
Creating relationships between documents 236

16 Maintaining Content 241
Flow of operations 241
Previewing content 242
Editing content 244
Modifying properties 245

Assigning a document's folder, categories, and taxonomies 246
Modifying display styles 249
Editing document types 251
Editing document fields 252
Setting document expiration dates 252
Deleting content 253
Deleting folders 253
Deleting taxonomies and categories 254
Deleting documents 254
Deleting display styles 255
Deleting document types 255
Deleting and removing document fields 257

17 Administering Content 259
About content administration 259
Flow of operations 260
Checking documents in and out 260
What happens during checkout 261
What happens during checkin 263
Checkin and checkout procedures 263
Administering version control 264

18 Searching Content 271
Setting up the CMS Administration Console search facility 271
Using the search facility in the CMS Administration Console 272
Search options 274

19 Managing Content Security 279
About content security 279
Flow of operations 280
Permissions for content access 281
User permissions required for CM operations 282
Cascading security 283
Setting security on CM elements 284

20 Importing and Exporting Content 287
About the import and export facilities 287
Summary of CMS Administration Console import and export behavior 288
Exporting content 289
Exporting from the toolbar 289
Exporting from a Property Inspector 292
Customizing exports 293
Importing content 293
Configuring the import process 294
Importing from the toolbar 294
Importing from a Property Inspector 295
Structure of the data import or export archive 297

Best practices and prerequisites 298
Planning for large-scale import/export operations 298
Security considerations 299

21 Administering Automated Tasks 301
The task display 302
Starting and stopping tasks 303

PART IV APPLICATIONS 305

22 Content Query Application 307
About Content Query 307
Using the Content Query action 308

PART V REFERENCE 313

23 Content Management Tag Library 315
Alphabetical list of tags 316
checkIn 316
checkOut 317
findDocuments 318
getChildDocuments 321
getContent 322
getDirectory 323
getDirectoryList 325
getDocType 329
getDocument 330
getFieldInfo 331
getFields 332
getLinkedDocuments 334
getVersionHistory 335
publish 336
unCheckOut 338
updateDocument 338

About This Book

Purpose

This book shows how to use the Content Management (CM) subsystem of Novell® exteNd Director™.

Audience

This book is for anyone who creates, manages, and accesses content in the CM subsystem, whether via the CM API or the CMS Administration Console.

Prerequisites

This book assumes you are familiar with the Java programming language, the Internet, and Web applications.

Learning materials on these topics are readily available from a variety of public and commercial sources.

Concepts

Describes the fundamentals of the Content Management (CM) subsystem and API programming

- Chapter 1, “About the Content Management Subsystem”
- Chapter 2, “Developing Content Management Infrastructure”
- Chapter 3, “Managing Documents”
- Chapter 4, “Securing Content”
- Chapter 5, “Managing Tasks”
- Chapter 6, “Managing Content Caching”
- Chapter 7, “Importing and Exporting Content”
- Chapter 8, “Working with Content Management Events”

1 About the Content Management Subsystem

This chapter provides an overview of the Content Management (CM) subsystem and includes the following topics:

- About content management
- Subsystem infrastructure
- Content life cycle
- Subsystem support functions
- Integration with other subsystems

About content management

The CM subsystem provides a repository for documents, enabling you to create and version documents, manage document security, search the repository, and so on. The CM subsystem provides Web CM capabilities such as style and layout management and document publishing and expiration.

The CM API and CMS Administration console provide interfaces to the CM subsystem that assist you in managing Web content. Other front-end applications can use the CM subsystem as a general document management system. For example, you could use a WebDAV application and the CM subsystem to manage CAD files or legal documents.

About content

What is meant by *content*? Content is defined as information that is viewed or downloaded by users of your exteNd Director application. The content managed by the CM subsystem is retrieved dynamically for online viewing or downloading when end users access your exteNd Director application.

The CM subsystem can store any type of content that can be digitized. It might store:

- Text documents, with XML or HTML tagging or in any word processing format
- Image files, such as GIF, JPG, QuickTime, and any other format
- Sound files
- Executable files
- Any other type of binary data

You can also store documents that support your content, such as:

- XSL style sheets
- XML DTDs
- Other content resources

It is up to you to store content in formats that are appropriate to your online application. A document doesn't have to be a complete item that would be displayed as is. A document can be a piece of data that you want to combine with other documents before displaying it, or some code resource that allows you to get data. For example, a document's content could be an URL, a set of URLs, a SQL statement, a paragraph, or an image.

About documents

The center of the CM subsystem is the *document*. Each document is described by a set of *metadata* that is a definition or description of data—in other words, data about data. In the CM subsystem, a document consists of all information required to maintain content (including the document's metadata, content, and versions) and all specifications for categorization, display characteristics, linked documents, access control, and so on.

A *checkout*/*checkin* system protects documents while you are changing them, and *versioning* allows you to maintain a history of content changes.

Publishing a document lets you choose a particular version of the document's content to make public. Once a version is published, you can define a fixed lifetime after which the version *expires* and can be archived and deleted.

📖 For more information, see Chapter 3, "Managing Documents".

Content and pages

It is important to distinguish the type of content managed by the CM subsystem from the *pages* managed by the Portal subsystem. Pages constitute the structure of the application, defining the graphical user interface (GUI) that helps users navigate the site. Pages contain *portlets*—the building blocks of an exteNd Director portal application. It is within portlets that application developers write code to search for and retrieve content managed by the CM subsystem in response to rules and real-time user interactions. Typically, pages change infrequently—while content is more dynamic.

The CM subsystem enables you to manage content structure, display style, versioning, categorization, and security to facilitate the retrieval—and preserve the integrity—of information presented to end users of your application. The Portal subsystem manages the actual application, including the interface and architecture in which this content is presented.

📖 For more information about pages and the Portal subsystem, see the section on portal concepts in the *Portal Guide*.

Subsystem infrastructure

The CM subsystem infrastructure establishes the criteria for organizing, displaying, managing, and securing your content. It is designed to support the basic unit of content—the *document*.

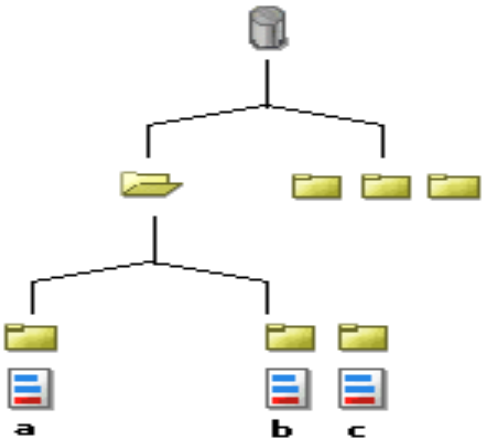
There are two levels of infrastructure: physical and logical. You must set up the physical infrastructure before you can create documents. Optionally, you can also define a logical infrastructure anytime.

Physical infrastructure

The physical infrastructure organizes the storage of documents in physical memory. This infrastructure consists of these components:

Component	Description
	Root folder
	Folder
	Document

There is a hierarchical relationship between folders and documents:



The top-level container is the *root folder*, which can contain one or more folders. A root folder is essentially just a specialized type of folder, one with no parent. In turn, *folders* can contain one or more *documents* or other folders. Each document resides in one (and only one) folder.

Logical infrastructure

The logical infrastructure organizes documents into logical groupings that can be used to provide a user's view of content. There are several elements:

Element	Description	Required or optional?
Field	Extension metadata content that can be shared by multiple documents. Documents can have one field, multiple fields—or none at all.	Optional
Document type	The basic classification mechanism for documents. Document types act as templates and provide groupings of fields. Every document must be associated with a document type. The CM subsystem attaches a default document type to all documents, but you can override this default.	Required
Display style	A classification for the look and feel of a document. This is sometimes called a *layout style*. Every document type can be associated with a display style for which you can define application-specific XML specifications for rendering documents uniquely for particular user agents. The CM subsystem attaches a default display style to all document types, but you can override this default.	Optional
Taxonomy	A classification system often used in Web portal design to describe categories and subcategories of content found on a Web site. Documents do not need to be classified under a taxonomy.	
Category	A descriptive name used to group documents logically. Documents do not need to be categorized.	

📖 Document types, fields, and display styles define the structure and layout of documents, as described in "Defining content structure and layout" on page 24. Taxonomies and categories classify documents for search and retrieval, as described in "Classifying content" on page 25.

Defining content structure and layout

Before you create documents, the structure of the content must be defined. Before you publish documents, the look and feel of the content must be defined to determine how the information will appear to users of the Web site. Typically, a content administrator oversees these tasks by developing the fields, document types, display styles, folders, and categories described under "Logical infrastructure" above.

Content developers associate document types and display styles with the documents they create by following this pattern:

1. Create a document type.
2. Create an instance of the document type and then create an XSL style sheet based on the content of that document.
3. Upload this XSL style sheet into a display style defined for the document type.

All documents you create based on the document type will contain the content structure and layout defined in the document type's display style.

Document types

A content administrator can create any number of document types, which consist of fields of information that dictate the structure of documents.

The CM subsystem provides default document types that can be accessed and modified by content administrators. In the CM API, the default document type is called **Default**; in the CMS Administration Console, it is called **_PmcSystemDefaultType**. These document types can be used to enforce a corporate standard for content or to create content in the absence of any custom document types.

Display styles

The CM subsystem comes with a default display style that is applied to all document types unless you override it with custom display styles.

Content administrators can define custom display styles that use one or more XSL style sheets developed in external editors and then uploaded to the CMS Administration Console. Each XSL style sheet specifies how to render content for a particular user agent, such as Microsoft Internet Explorer and Netscape Navigator.

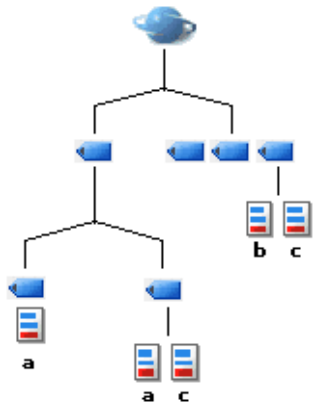
When you have specified your display styles appropriately, the CM subsystem automatically matches the desired style to the user agent that is active in real time.

Classifying content

You can create content without classifying it; but if your exteNd Director application allows users to select categories of information, a content administrator may want to create categories for grouping documents in a logical fashion. That way portlets can more easily access the documents specified by users as they interact with the Web site.

For example, suppose an exteNd Director application developer creates a portlet that lists URLs that link to specific documents. If documents are classified by category, the portlet can link to all documents of a particular category by looking for a parameter called **category** passed on the URL.

There is a hierarchical relationship between categories and documents:



The top-level container is the *root category*, which can contain one or more categories. In turn, a *category* can contain one or more *documents* or other categories. A single document can be associated with any number of categories—or with no categories at all.

Content life cycle

The CM subsystem maintains a history of all document changes. The version history for a document might look like this:

Version ID	MIME type	Content data	Size	Date	Modifier	Comment
3	text/html	[content v3]	47K	6/16/00	bbrown	New facts
2	text/html	[content v2]	45K	6/11/00	bbrown	Fleshed-out content
1	text/html	[content v1]	24K	6/10/00	ssmith	Created

Checking out documents

When you check out a document, it becomes locked; no one else can check it out until you check it in or cancel the checkout. (Exception: the CM subsystem allows administrators to remove locks on documents if that becomes necessary.)

When you check in a document whose content has changed, a new version of that content is created. If you change the metadata but not the content, no new version is created when you check in the document. The metadata is updated but not versioned.

Publishing a document

When a document has been approved, its content can be published as the officially released version of the document. When an application requests a document, the published version is the one provided.

The published version is not necessarily the latest one, however. Modifications can continue as content developers check out the most recent version of the document. Publishing a document creates a stable version of the document for the public.

Subsystem support functions

The CM subsystem includes built-in support for these functions:

Function	Description	For more information see
Content caching	CM function that allows you to configure caching for different elements	Chapter 6, “Managing Content Caching”
Task management	CM function for configuring background execution of specific operations such as publishing documents	Chapter 5, “Managing Tasks”
Content import and export	Facilities for importing and exporting content in and out of the CM subsystem	Chapter 7, “Importing and Exporting Content”

Integration with other subsystems

You can integrate CM with any other exteNd Director subsystem, including:

Related subsystem	Description	For more information see
Search	Supports conceptual and keyword searching of document content and metadata.	Chapter on conceptual searching in the *Content Search Guide*
Security	Used to secure access to CM subsystem elements.	Chapter 4, “Securing Content”
Workflow	Used to access CM documents in workflow applications.	Chapter on the Content Life Cycle application in the *Workflow Guide*

2 Developing Content Management Infrastructure

This chapter describes how to set up and manage the infrastructure for the Content Management (CM) subsystem using the CM API. It has these sections:

- About the CM API
- About the CM subsystem infrastructure
- Managing fields
- Managing document types
- Managing layout styles
- Managing folders and categories
- Navigating the CM hierarchy

NOTE: This chapter describes an exteNd Director API that allows you to build your own CM application. exteNd Director also provides the CMS Administration Console, which you can use to create, maintain, administer, and secure all content for your exteNd Director application.

📖 For more information, see Chapter 13, "Setting Up the Required Infrastructure" and Chapter 14, "Setting Up the Optional Infrastructure".

About the CM API

You can use the CM API to build a system tailored to your business process. By writing portlets, you can build a complete interface that includes such functionality as:

- Adding and managing documents
- Checking out documents for editing
- Versioning documents
- Approving document versions for publication
- Building layout styles for XML content
- Providing comprehensive searching functionality of content and metadata
- Providing security for content objects

The CM API provides complete programmatic access to the document repository.

Getting a content manager object

Methods of the EbiContentMgmtDelegate interface provide access to the most of the objects in the CM subsystem.

For all the examples in this chapter, you must use this code somewhere in your portlet to get a reference to the content manager delegate:

```
EbiContentMgmtDelegate defaultCmgr =
    com.sssw.cm.client.EboFactory.getDefaultContentMgmtDelegate();
if (cmgr != null)
   ... // do content-related processing
else
   System.out.println("Failed to get Content Manager");
```

Using delegates *Delegates* are objects that provide a layer of abstraction for main exteNd Director manager objects (such as the Content Manager object). Using delegates removes the need for coding things like local and remote access to exteNd Director services.

From a best-practices standpoint, you should always use delegates rather than accessing exteNd Director manager objects directly.

Changing repository data

In the simplest case, the basic procedure for working with objects in the repository is:

1. Use a get method of EbiContentMgmtDelegate to get an object from the repository.
2. Use methods of that object to modify it.

3 Use an update method of EbiContentMgmtDelegate to put the changed object back in the repository, or use the update method on the object itself if it is available.

Some objects are more complex. The rest of this chapter describes how to work with many of these objects, with code examples.

About the CM subsystem infrastructure

Before creating documents in the CM subsystem, you must set up the content infrastructure, which includes the criteria by which you organize the documents. The infrastructure includes fields, document types, layout styles, folders, and categories:

Item	Description	For more information
Fields	A field allows you to provide application-specific information about documents, also called *extension metadata*. Each document type can have zero or more fields. Each document may have one or more values per field, and null values are allowed.	"Managing fields" on page 32
Document types	The document type is the basic classification mechanism of the system. You would classify documents as a particular type when they have similar formatting and subject matter. A document type has a list of fields and a default layout style.	"Managing document types" on page 35
Layout styles	A document type can have a default layout style. Specific documents can have their own layout styles or sets of styles.	"Managing layout styles" on page 38
Folders	Folders allow you to group documents for administrative purposes. For example, you can assign confidential documents to a folder that has restricted access. Folders can be nested.	"Managing folders and categories" on page 43
Categories	You can use categories as another way of organizing documents. Typically, categories are the user's view of the content repository, organized by subject matter. Categories can be nested.	"Managing folders and categories" on page 43

Managing fields

All documents have a basic set of metadata, such as title, author, abstract, published version, and so on. You can also define custom metadata fields to store application-specific data for each document type. Fields are appropriate for any piece of data for which all the documents have a value. For example, movie reviews have a director, cast, release date, and rating. Books have an author, publisher, publish date, and number of pages. Reviews of travel destinations have country, cost category, and quality rating.

Fields are also useful for finding documents. For each document type, a set of fields identify the pertinent, searchable information for the subject matter of that document type. Fields can be searched quickly via a database lookup, in contrast to searching the document content text.

For example, for a document type of MovieReview, you might create several fields as shown below:

Field name	Data type	Sample value
Genre	FT_STRING	Drama, Romance
Tagline	FT_STRING	In a perfect world...they never would have met
User Rating	FT_STRING	4.9/10 (1083 votes)
Runtime	FT_STRING	USA:133 / UK:132 / Finland:133 / Japan:132
Year of Release	FT_INT	2000

NOTE: In this example, Genre and Runtime could have multiple values.

Data types EbiDocField defines several data types to be used for fields. This table categorizes the available types:

Type of data	Available data types defined in EbiDocField
Character data	FT_CHAR, FT_STRING
Numeric	FT_BIGDECIMAL FT_DOUBLE, FT_FLOAT FT_INT, FT_LONG, FT_SHORT
Boolean	FT_BOOLEAN
Date and time	FT_DATE, FT_TIME, FT_TIMESTAMP
Binary	FT_BYTE, FT_BYTEARRAY

Metadata for fields You already know that fields store metadata about a document. You can also store data about the field itself. You can use this *extension metadata* to store a list of appropriate values, a prompt to use in forms, an image for the field, or other information appropriate to your application. The data is a byte array.

Fields and document types When you create a document type, you specify the set of fields it uses. You can use a field with more than one document type.

Fields and values For each document of a particular document type, all the associated fields must have at least one value, specified via an EbiDocExtnMetaInfo object. The value can be null. You assign the field values to the document as a set via an EbiDocExtnMeta object. EbiDocExtnMeta holds an EbiDocExtnMetaInfo object for each field associated with the document type. You call getFieldValues() to get an array of values for a field. The values can be returned as Strings, or they can have the field's data type.

These methods in EbiContentMgmtDelegate let you add and modify fields:

Method	Returns	Description
addDocumentField()	EbiDocField	Adds a field to the CM subsystem. You specify the name, data type, supporting data for the field, and an *ACL* (access control list). The last two arguments can be null.
getDocumentFieldByID()	EbiDocField	Gets a field by ID.
getDocumentFieldByName()	EbiDocField	Gets a field by name.
updateDocumentField()	void	After calling methods to modify an EbiDocField object, updates the content repository with the changes.
removeDocumentField()	void	Removes a field from the system.
getDocumentFields() and getFilteredDocumentFields()	Collection of EbiDocField	Gets a Collection of all the fields in the CM subsystem. The filtered version omits fields to which the current user has no READ access. The unfiltered version gets all fields, regardless of access rights.

📖 For information about using fields with document types, see "Managing document types" on page 35.

Adding a field

This example provides a method called **addField()** that adds an extension metadata field:

```
public void addField(EbiContentMgmtDelegate cmgr, EbiContext
context)
        throws EboUnrecoverableSystemException,
EboSecurityException, EboItemExistenceException
    {
        String fieldName = "Rating";
        String valueType = EbiDocField.FT_STRING;
        String extnMeta = "This is a Rating field...";
        cmgr.addDocumentField(
            context,                  // Context
            fieldName,                // Field name
            valueType,                // Value data type
            extnMeta.getBytes(),      // Extension metadata
            null);                    // ACL
    }
```

Adding a field to a portlet

This example shows how to add a field to a portlet's processAction(0 method. It gets the name and data type the user entered in an HTML form and adds a field. A message about success or failure is stored in the context object to be displayed when the portlet content is generated.

```
public void processAction (ActionRequest request, ActionResponse
response){

   String name = request.getParameter(FORM_NAME);
   String datatype = request.getParameter(FORM_DATATYPE);
   String valuelist = request.getParameter(FORM_LIST);

   EbiContentMgmtDelegate cmgr = ...; // get content manager

   try
   {
     cmgr.addDocumentField(context,name,datatype,valuelist,null);
     context.setValue(
         this.getPortletName() + KEY_STATUS,
         "Field " + name + " successfully added.");
   }
   catch (Exception e)
   {
     context.setValue(
         this.getPortletName() + KEY_STATUS,
         "Field " + name + " not added.");
   }
}
```

Listing fields using different filters

This example provides a method called **listFields()** that gets existing document fields by filtering the results in different ways.

The listFields() method needs to have access to a content manager (EbiContentMgmtDelegate) and context object (EbiContext), which are passed in as arguments. The context object provides information about the user's security privileges. The listFields() method passes the context object to the getFilteredDocumentFields() method to return only those fields for which the user has READ access:

```
public void listFields(EbiContentMgmtDelegate cmgr, EbiContext context)
        throws EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException
    {
        // Get all the existing fields (note: no security checking is done here)
        Collection allFields = cmgr.getDocumentFields(context);
        Iterator iterAllFields = allFields.iterator();
        while (iterAllFields.hasNext())
        {
            EbiDocField field = (EbiDocField)iterAllFields.next();
            System.out.println(field + "\n\n");
        }

        // Get all the fields that belong to doctype 'MovieReview'
        EbiDocType docType = cmgr.getDocumentTypeByName(context,"MovieReview");
        Collection docTypeFields = cmgr.getDocumentFields(context,docType.getDocTypeID());

        // Get all the fields to which the user has Read access
        Collection filteredFields = cmgr.getFilteredDocumentFields(context);

        // Get all the Read-accessible fields that belong to doctype 'MovieReview'
        Collection filteredDtFields = cmgr.getFilteredDocumentFields(context,
docType.getDocTypeID());
    }
```

Managing document types

A document type identifies a particular type of content. Typically, you create document types for groups of documents that have similar content. The documents share the same set of fields that describe that content and, for XML content, the same layout styles to display the content.

After you have created a document type, you can modify its name and description. To do so, get an EbiDocType object, call setDocTypeName() or setDescription(), then call updateDocumentType() to put the changed type back into the content repository.

TIP: You can also associate layout styles with the document type. For information, see "Managing layout styles" on page 38.

These methods in EbiContentMgmtDelegate let you **add and modify document types**:

Method	Returns	Description
addDocumentType()	EbiDocType	Adds a document type to the system. You specify a name, description, and the list of metadata fields associated with the type. The system gives the type a numeric ID.
getDocumentType()	EbiDocType	Gets a document type by name or ID.
updateDocumentType()	void	After calling methods to modify an EbiDocType object, updates the content repository with the changes.
removeDocumentType()	boolean	Removes a document type from the system. If documents of that type exist, you must delete them before you can delete the type.
getDocumentTypes() and getFilteredDocumentTypes()	Collection of EbiDocType	Gets a Collection of EbiDocType objects. The filtered version omits types to which the current user has no READ access. The unfiltered version gets all types, regardless of access rights.

These methods of EbiContentMgmtDelegate **manage the association between document types and fields**:

Method	Returns	Description
addFieldToDocumentType()	void	Adds a field to the document type. For existing documents, the values for the field are null.
removeFieldFrom-DocumentType()	boolean	Removes the association between a field and a document type. Deletes the field values for documents of that type.
getDocumentFields() and getFilteredDocumentFields()	Collection of EbiDocField	Gets the document fields for a document type. The filtered version omits fields to which the current user has no READ access. The unfiltered version gets all fields for the type, regardless of access rights.
getDocumentTypesWithField() and getFilteredDocument-TypesWithField()	Collection of EbiDocType	Gets a Collection of all the document types that use a particular field.

Adding a document type with associated fields

This example provides a method called **addDocType()** that adds a document type called **Movie Review** and associates it with several existing fields. The addDocType() method needs to have access to a content manager (EbiContentMgmtDelegate) and context object (EbiContext), which are passed in as arguments.

```
public void addDocType(EbiContentMgmtDelegate cmgr, EbiContext context)
        throws EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException
    {
        // Get several fields by name
        EbiDocField fldDir = cmgr.getDocumentFieldByName(context, "Director");
        EbiDocField fldGenre = cmgr.getDocumentFieldByName(context, "Genre");
        EbiDocField fldYear = cmgr.getDocumentFieldByName(context, "Year");
        EbiDocField fldCast = cmgr.getDocumentFieldByName(context, "Cast");
        // Get the field IDs
        String[] fieldIDs = {
            fldDir.getFieldID(),
            fldGenre.getFieldID(),
            fldYear.getFieldID(),
            fldCast.getFieldID() };
```

```
        // Add the doctype
        EbiDocType dt = cmgr.addDocumentType(
            context,                          // Context
            "Movie Review",                   // Doctype name
            "Movie Review document type",     // Description
            fieldIDs,                         // Associated fields
            null);                            // ACL for the doctype
        System.out.println("The new doctype: " + dt);
    }
```

Managing layout styles

Layouts are XSL specifications for rendering a document. The document might be XML or some other format that can be processed by XSL. The actual layout specification is stored as the content of a document in the repository. The CM subsystem has a document type called **Document Layout** already installed for layout documents. You can use it or add your own document types for layouts.

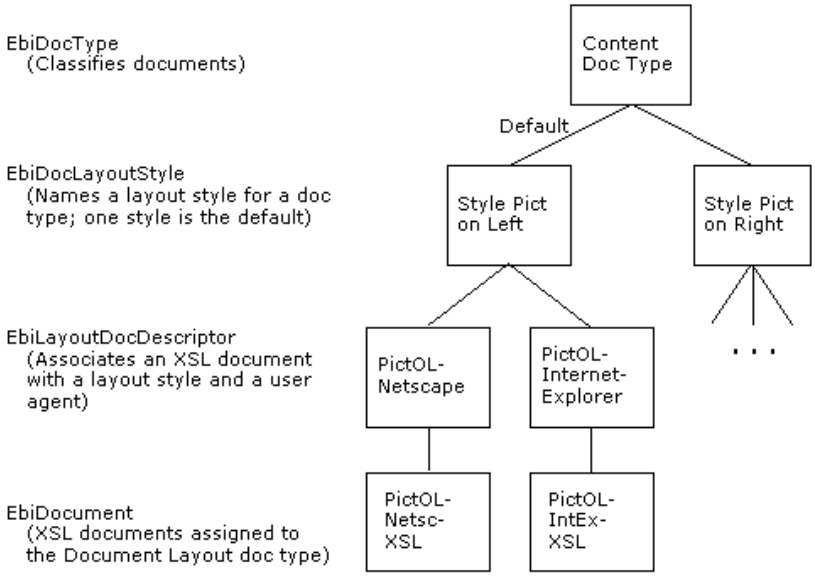
What you can do After you have added a layout document, you can check it out, modify it, and check it in. That means a particular layout document can have multiple versions. You can publish one of those versions.

You can group several layouts together under the umbrella of a layout style. The various layouts in the layout style can handle the rendering of the document for different clients (also called *user agents*), such as browsers, PDAs, and other display devices. The association of a layout document with a user agent is handled by a layout document descriptor.

Layout styles and document types A layout style is associated with a document type. When you display a document of that type, the system searches the layout document descriptors in the style to find the one for the user agent, as specified in the portlet's context object.

A layout style with multiple layout document descriptors can process content for various clients. When you want to display a document of the particular document type, you call getDocumentLayout(); the system gets the current user agent from the context object to select the appropriate layout.

Here is the group of objects that provide XSL processing for a content document:



NOTE: In addition to layout styles for document types, you can define a **layout set** for a specific document. A layout set is a custom combination of layout documents for a single content document. This specialized functionality is appropriate for special types of documents. When you are producing many documents of the same type, you will typically stick with layout styles for the document type. For more information, see "Specifying layout sets for documents" on page 60.

➢ **To set up layout styles for a document type:**

1. Add one or more layout styles for the content document type.
2. Specify one of the styles as the default for that document type.
3. Add one or more layout documents whose XSL is designed for the expected content. The versions can arrange the content differently or tailor the content for different clients.
4. Add layout document descriptors that tie the layout documents to a client and a layout style.

These methods in EbiContentMgmtDelegate let you **add and modify layout styles and their associated objects**:

Method	Returns	Description
addDocumentLayoutStyle()	EbiDoc-LayoutStyle	Adds a new Document Layout Style for the specified Document Type.
getDocumentLayoutStyle()	EbiDoc-LayoutStyle	Gets the details of a particular layout style.
updateDocumentLayout-Style()	void	Updates the information for a layout style in the CM subsystem.
removeDocumentLayout-Style()	boolean	Removes a layout style from the system.
addLayoutDocument-Descriptor()	EbiLayout-DocDescriptor	Adds a layout document descriptor, associating a layout document with a layout style and user agent.
getLayoutDocument-Descriptor()	EbiLayout-DocDescriptor	GetS a layout document descriptor object.
updateLayoutDocument-Descriptor()	void	Updates a layout document descriptor with a new user agent.
removeLayoutDocument-Descriptor()	boolean	Removes a layout document descriptor.
getLayoutDocument-Descriptors()	Collection of EbiLayout-DocDescriptor	Gets the layout document descriptors associated with a layout style.
getDocumentLayout()	EbiDoc-Version-Descriptor	Gets the layout document appropriate for the current document and user agent. This is the actual XSL you use to process the content document.
getDefaultDocument-LayoutStyle()	EbiDoc-LayoutStyle	Gets the layout style that is the default for a document type.
getDocumentLayout-Styles() and getFiltered-DocumentLayoutStyles()	Collection of EbiDoc-LayoutStyle	Gets all the layout styles associated with a document type. The filtered version omits styles to which the current user has no READ access. The unfiltered version gets all styles for the type, regardless of access rights.

User agents

A user agent identifies itself in the HTTP header it sends to the server. exteNd Director stores the identifying string in the context object. The string used by a browser varies according to the browser version. Here are some examples:

```
User Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98)
User Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT)
User Agent: Mozilla/4.5 (Macintosh; U; PPC)
User Agent: Mozilla/4.7 [en] (WinNT; I)
User Agent: Mozilla/3.0 (compatible; Opera/3.0; Windows 95/NT) 3.1
```

You will need to use these strings in EbiLayoutDocumentDescriptor objects.

For more information on user agents, see the HTTP 1.1 specification.

Adding a layout style

This example provides a method called **addLayoutStyle()** that adds a layout style for a document type called **Movie Review**. The addLayoutStyle() method needs to have access to a content manager (EbiContentMgmtDelegate) and context object (EbiContext), which are passed in as arguments:

```
public void addLayoutStyle(EbiContentMgmtDelegate cmgr, EbiContext context)
        throws EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException
    {
        // Get the doctype for which the style is to be added
        EbiDocType dtMovieReviews = cmgr.getDocumentTypeByName(context, "Movie Review");

        // Add the new style
        EbiDocLayoutStyle style = cmgr.addDocumentLayoutStyle(
            context,                                          // Context
            dtMovieReviews.getDocTypeID(),                    // Doctype ID
            "MovieReviewStyle-PicOnLeft",                     // Style name
            "Layout style for movie reviews, with pic on left",  // Style descr
            true,                                             // Is default style
            null);                                            // ACL for style
        System.out.println("The new style: " + style);
    }
```

Adding a layout document and a layout document descriptor

This example provides a method called **addLayoutDocAndDescriptor()** that adds a layout document and a layout descriptor. The layout descriptor associates the layout document with the layout style from the previous example. The addLayoutDocAndDescriptor() method needs to have access to a content manager (EbiContentMgmtDelegate), context object (EbiContext), layout file name, and layout style, which are passed in as arguments:

```
public void addLayoutDocAndDescriptor(
        EbiContentMgmtDelegate cmgr, EbiContext context, String layoutFileName, String
layoutStyleID)
        throws
            EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException, FileNotFoundException, IOException
    {
        // Read in the XSL for the layout
        FileInputStream fis = new FileInputStream(layoutFileName);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        byte[] value = new byte[4096];
        while (true)
        {
            int bytes = fis.read(value);
            if (bytes < 1)
                break;
            baos.write(value, 0, bytes);
        }
        byte[] content = baos.toByteArray();
        baos.close();

        // Get the document layout doctype
        EbiDocType dtLayout = cmgr.getDocumentTypeByName(context, "Document Layout");
        // Get the Layouts folder
        EbiDocFolder layoutFolder = (EbiDocFolder)cmgr.lookupDirectoryEntry(
            context, "MyApp/Layouts", EbiDocFolder.EL_DOC_FOLDER);

        // Add the layout document
        EbiAddDocumentParams params = cmgr.createAddDocumentParams();
        params.setName("ReviewLayout-POL");
        params.setDocTypeID(dtLayout.getDocTypeID());
        params.setFolderID(layoutFolder.getID());
        params.setAuthor("JSmith");
        params.setTitle("ReviewLayout-POL");
        params.setSubtitle("This is the layout with picture on left");
        params.setMimeType("text/xsl");
        params.setContent(content);
        params.setComment("Initial revision.");
        // params.setAcl(...); specify an ACL, otherwise inherit ACL of parent folder
        EbiDocument layoutDoc = cmgr.addDocument(context, params);
        System.out.println("New layout doc: " + layoutDoc);

        // Publish the new layout document
        cmgr.publishDocumentContentVersion(context, layoutDoc.getID(), 1, true, true);
```

```
        // Figure out what user agent this layout is intended for
        String userAgent = "User Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT)";

        // Associate the new layout document with the specified layout style
        EbiLayoutDocDescriptor ldd = cmgr.addLayoutDocumentDescriptor(
            context,               // Context
            layoutStyleID,         // Layout style ID
            layoutDoc.getID(),     // Layout document ID
            userAgent);            // User agent
    }
```

Changing a layout style

This example presents a method called **changeLayoutStyle()** that gets the default style for a document type and changes it so that it is not the default. The changeLayoutStyle() method needs to have access to a content manager (EbiContentMgmtDelegate) and context object (EbiContext), which are passed in as arguments:

```
public void changeLayoutStyle(EbiContentMgmtDelegate cmgr, EbiContext context)
        throws EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException
    {
        EbiDocType dtMovieReview = cmgr.getDocumentTypeByName(context, "MovieReview");
        EbiDocLayoutStyle style = cmgr.getDefaultDocumentLayoutStyle(context,
dtMovieReview.getDocTypeID());
        style.setDefault(false);
        cmgr.updateDocumentLayoutStyle(context, style);
    }
```

Managing folders and categories

Folders and categories are ways of organizing documents. A document belongs to one folder and can belong to many categories. Typically, you would use folders to group documents for administrative purposes, such as all documents for a project or documents that have access restrictions. You can use categories to organize documents as an end user might view them, typically by subject matter.

The system has a root folder and root category already created—called Root Folder and Root Category. The content manager provides the getRootFolder() and getRootCategory() methods to get EbiDocFolder and EbiDocCategory objects for them.

The default directory type for folders and categories is EbiDirectory.DIR_TYPE_DEFAULT. The root and system types apply to the root folder and root category. You can also define your own folder types. For information, see EbiDirectory in the *API Reference*.

These methods of EbiContentMgmtDelegate let you **manage folders and categories**:

Method	Returns	Description
addFolder()	EbiDocFolder	Creates a new folder.
copyFolder()	EbiDocFolder	Copies one folder into another.
getFolder()	EbiDocFolder	Gets a folder by name or ID.
moveFolder()	EbiDocFolder	Moves one folder into another.
updateFolder()	void	Updates a folder in the content repository after making changes to its properties via the EbiDocFolder object.
removeFolder()	boolean	Removes a folder. If the folder contains documents and subfolders, you can set the force argument to remove them too. The user must have WRITE permissions on all the subfolders and documents; otherwise, a security exception is thrown. If force is false, the folder can't be removed until the contents are deleted.
getRootFolder()	EbiDocFolder	Gets the top-level folder.
addCategory()	EbiDocCategory	Creates a new category.
copyCategory()	EbiDocCategory	Copies one category into another.
getCategory()	EbiDocCategory	Gets a category by name or ID.
moveCategory()	EbiDocCategory	Moves one category into another.
updateCategory()	void	Updates a category in the content repository after making changes to its properties via the EbiDocCategory object.
removeCategory()	boolean	Removes a category.
getRootCategory()	EbiDocCategory	Gets the top-level category.

Method	Returns	Description
addDocument-CategoryReference()	void	Adds a document to a category.
removeDocument-CategoryReference()	boolean	Removes a document from the category.
getDocumentCategory-References() and getFilteredDocument-CategoryReferences()	Collection of EbiDocCategory	Gets the categories to which the document belongs. The filtered version omits categories to which the current user has no READ access. The unfiltered version gets all categories for the document, regardless of access rights.

Adding a category

This example presents a method called **addCategory()** that gets the information required for creating a new category, then adds the new category as a subcategory of the specified parent. The addCategory() method needs to access a content manager (EbiContentMgmtDelegate) and context object (EbiContext), which are passed in as arguments:

```
public void addCategory(EbiContentMgmtDelegate cmgr, EbiContext context)
        throws EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException
    {
        // Locate the parent category
        EbiDocCategory categParent = (EbiDocCategory)cmgr.lookupDirectoryEntry(
            context, "MyApp/Shopping", EbiDocCategory.EL_DOC_CATEGORY);
        EbiDocCategory categChild = cmgr.addCategory(
            context,                            // Context
            categParent,                        // Parent category
            "Clothing",                         // Tew category name
            EbiDirectory.DIR_TYPE_DEFAULT,      // type of the new category
            "This is the clothing-related category", // Description
            null);                              // ACL for the new category
        System.out.println("New category added: " + categChild);
```

Navigating the CM hierarchy

Once your directory hierarchy is established, you can get a listing of the contents of a directory and examine the properties of individual entries.

This section describes some ways to use the methods and classes that navigate the directory hierarchy. Both categories and folders implement the functionality for directory manipulation found in their superinterface EbiDirectory. Folders, categories, and documents also implement EbiDirectoryEntry and share methods for getting information about the contents of a directory.

Methods These methods are useful in navigating categories and folders:

- **getRootCategory()** and **getRootFolder()** of EbiContentMgmtDelegate get the top of a directory hierarchy.
- **getDirectoryList()** and **getFilteredDirectoryList()** of EbiContentMgmtDelegate return a collection of EbiDirectoryEntry objects. You can specify whether the list includes subdirectories, documents, or both.
- **isDirectory()** of EbiDirectoryEntry reports whether an entry is a directory or a document.
- **lookupDirectoryEntry()** of EbiContentMgmtDelegate gets an EbiDirectoryEntry object for a category, folder, or document based on a path built from the names of the parent objects in the hierarchy.
- **getEntry()** of EbiContentMgmtDelegate gets an entry by name in the specified directory.

Example This example builds an XML DOM tree of nested categories, starting with the root category. The root category is a category element within Categories; subcategories of the root and further nested levels are category elements also. The name and ID for each category are attributes.

The code creates the Categories container element and gets the root category of the tree you want to build. It then calls addNode() to find and add its subcategories. The variable **dom** is the DOM object and **root** is the root element of the DOM.

```
Element categories = dom.createElement("Categories");
root.appendChild(categories);

EbiDocCategory category = cmgr.getRootCategory(context);
if (category == null)
   System.out.println("root category is null");
else
{
   Element rootCategory = dom.createElement("category");
   categories.appendChild(rootCategory);
   rootCategory.setAttribute("id", category.getID());
   rootCategory.setAttribute("name", category.getName());
   addNode(rootCategory, category, dom, context,
      cmgr, "category");
}
```

The addNode() method gets the subcategories of a particular category and adds them as child elements. It is called recursively to add additional levels of nested subcategories if they exist:

```
public void addNode(org.w3c.dom.Element element,
EbiDirectoryEntry directoryEntry, org.w3c.dom.Document document,
EbiContext context, EbiContentMgmtDelegate cmgr, String
elementName)
{
   try
   {
      Collection collection = cmgr.getFilteredDirectoryList(
         context, (EbiDirectory) directoryEntry, true, false);
      Enumeration list = Collections.enumeration(collection);
      if (list != null)
      {
         Element child;
         while (list.hasMoreElements())
         {
            EbiDirectoryEntry subdirEntry =
               (EbiDirectoryEntry) list.nextElement();
            child = document.createElement(elementName);
            child.setAttribute("id", subdirEntry.getID());
            child.setAttribute("name", subdirEntry.getName());
            element.appendChild(child);
            addNode(child, subdirEntry, document,
               context, cmgr, elementName);
         }
      }
   }
   catch (Exception e)
   {
      e.printStackTrace();
   }
}
```

3 Managing Documents

This chapter describes how to manage documents using the Content Management (CM) API. It has these sections:

- About documents
- Adding documents
- Specifying field values for a document
- Specifying layout sets for documents
- Creating links between documents
- Modifying and publishing documents
- Displaying documents

NOTE: Most of the document management tasks described in this chapter can also be accomplished using the CMS Administration Console.

 For more information, see Chapter 15, “Creating Content”. and Chapter 16, “Maintaining Content”.

About documents

A *document* in the CM subsystem may represent a simple, finite piece of content such as an image, or it may be a complex entity that comprises other documents. A document can be any data that you want to use directly or indirectly in your exteNd Director application.

The CM subsystem uses metadata fields to describe a document. There are standard fields for every document, such as name, title, author, and abstract. You can also associate content-related fields with a document type. This *extension metadata* can hold additional searchable information specific to that document type.

A document object can be associated with an EbiDocContent object that holds the text or binary data, but a document doesn't need to have a content object. The metadata for the document may store all the information you need. For a short text document, you could store the entire text in the abstract field. If the document doesn't have content, specify null for the MIME type and content.

The supplied content, if any, becomes the first version of the document. If you want to publish the content, you can call publishDocumentContentVersion() anytime or rely on your scheduled task to publish it. Documents without content cannot have versions (including a published version), but you could use another field (such as status) to label a document as publicly available.

Accessing the CM API

The EbiContentMgmtDelegate interface provides access to most of the document-related methods in the CM subsystem.

📖 For information, see "About the CM API" on page 30.

Adding documents

To add a document, you create an EbiAddDocumentParams object and set various parameters. The next table explains the default values for the required parameters: name, document type, folder, and extension metadata, if any—as well as other parameters for which the default value has a particular meaning. Any other metadata fields that aren't explicitly set are null:

Parameter	Description and default values
Name	A name for the document, used when specifying a path for the document in the folder structure. The default name is the UUID assigned to the document when it is added.
Document type ID	The ID of the document type for this document. The default is the system's Default document type.
Folder ID	The folder that contains this document. The default is the system's root folder.
Extension metadata	If the document belongs to a document type that has at least one associated extension metadata field, you must call the setExtensionMetaData() method to provide values for the fields. 📖 For information, see "Specifying field values for a document" on page 54.

Parameter	Description and default values
Publish date	A timestamp specifying when the document's current version should be published. The default value of null means publish as soon as possible.
Expiration date	A timestamp specifying when the document should be removed from the published area. The default value of null means never expire.
Access control list	An ACL specifying access rights to the document. The ACL is null by default. In this case, the document inherits the ACL of its folder. If the folder doesn't have an ACL, there are no restrictions for the document.

Adding a document

This code example presents a method called **addDocument()** that illustrates how to add a document of type **Movie Review**. This method sets all required document parameters—document type, name, title, author, and parent folder—as well as some optional parameters.

The new document does not contain extension metadata fields, nor does it have a parent document. The addDocument() method sets the content of the new movie review document explicitly and stores it in the byte array **content**.

The addDocument() method needs to access a content manager (EbiContentMgmtDelegate) and context object (EbiContext), which are passed in as arguments.

Note that the addDocument() method does not set the ACL for the new document. This means that the ACL is null and the document inherits the ACL of its folder:

```
public void addDocument(EbiContentMgmtDelegate cmgr, EbiContext context)
        throws EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException
    {
        // Get the doctype
        EbiDocType type = cmgr.getDocumentTypeByName(context, "Movie Review");

        // Get the folder
        EbiDocFolder folder = (EbiDocFolder)cmgr.lookupDirectoryEntry(
            context, "MyApp/MovieReviews/Current", EbiDocFolder.EL_DOC_FOLDER);

        // Get the content
        String movieContent = "This movie has exceeded all expectations!....";
        byte content[] = movieContent.getBytes();

        EbiAddDocumentParams docParams = cmgr.createAddDocumentParams();
        docParams.setName("Star Trek Movie Review");
```

```
        docParams.setDocTypeID(type.getDocTypeID());
        docParams.setFolderID(folder.getID());
        docParams.setAuthor("Night Ghost");
        docParams.setTitle("Star Trek Movie Review");
        docParams.setSubtitle("Generations");
        docParams.setAbstract("This reviewer loves the movie!........");
        docParams.setMimeType("text/xml");
        docParams.setContent(content);
        docParams.setComment("Initial revision.");

        // params.setAcl(...); specify an ACL, otherwise inherit ACL of parent folder

        EbiDocument doc = cmgr.addDocument(context, docParams);
        System.out.println("Added new movie review: " + doc);

        // Publish the new document
        cmgr.publishDocumentContentVersion(context, doc.getID(), 1, true, true);
    }
```

Adding multiple documents

This code example presents a method called **addMultipleDocuments()** that converts a set of files into new documents of type **Movie Review** and adds them to the CM subsystem.

This method sets all required document parameters—document type, name, title, author, and parent folder—as well as some optional parameters.

Note that the addMultipleDocuments() method executes the following shared logic outside the **for** loop for efficient processing:

- Sets the shared parameters **author**, **comment**, and **MIME type**
- Calls the **getDocTypeByName()** and **createAddDocumentParams()** methods

The new documents do not contain extension metadata fields, nor do they have parent documents. The addMultipleDocuments() method reads in the content of each new movie review from its file of origin and stores the data in the byte array **content**.

As for security, the addMultipleDocuments() method does not set the ACL for the new documents. This means that the ACL is null and the documents inherit the ACL of their folder.

The addMultipleDocuments() method needs to access a content manager (EbiContentMgmtDelegate), context object (EbiContext), and the directory where the files of interest are stored. All of these entities are passed in as arguments:

```
public void addMultipleDocuments(
        EbiContentMgmtDelegate cmgr, EbiContext context, String dirName)
        throws
            EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException,
```

```
        FileNotFoundException, IOException
{
    // Get the doctype
    EbiDocType type = cmgr.getDocumentTypeByName(context, "Movie Review");

    // Get the folder
    EbiDocFolder folder = (EbiDocFolder)cmgr.lookupDirectoryEntry(
        context, "MyApp/MovieReviews/Current", EbiDocFolder.EL_DOC_FOLDER);

    // Instantiate a document addition parameters object
    EbiAddDocumentParams docParams = cmgr.createAddDocumentParams();

    // Set all the String parameters to be reused
    String author = "NightGhost";
    String mimeType = "text/xml";
    String comment = "Initial revision.";

    File dir = new File(dirName);
    File[] files = null;
    if (dir.exists() && dir.isDirectory())
        files = dir.listFiles();
    else
        throw new EboApplicationException(null, "Invalid directory name '" + dirName
           + "'.");

    // Turn each file in the specified directory into a new movie review document
    for (int i = 0; i < files.length; i++)
    {
        if (files[i].isDirectory())
            continue;
        FileInputStream fis = new FileInputStream(files[i]);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        byte[] value = new byte[4096];
        while (true)
        {
            int bytes = fis.read(value);
            if (bytes < 1)
                break;
            baos.write(value, 0, bytes);
        }
        byte[] content = baos.toByteArray();
        baos.close();

        String name = files[i].getName();

        docParams.setName(name);
        docParams.setDocTypeID(type.getDocTypeID());
        docParams.setFolderID(folder.getID());
        docParams.setAuthor(author);
        docParams.setTitle(name);
        docParams.setMimeType(mimeType);
        docParams.setContent(content);
        docParams.setComment(comment);
        docParams.setPublishImmediately(true);
```

```
            // params.setAcl(...); specify an ACL, otherwise inherit ACL of parent folder

            EbiDocument doc = cmgr.addDocument(context, docParams);

        }
    }
```

Specifying field values for a document

When you add a document, you must create a set of field values that match the fields defined for the document's type. Each field can have one or more values, and null values are allowed. The fields and their values are called *extension metadata*, in contrast to the standard metadata defined for an EbiDocument object (such as title, author, abstract, and status).

You manage the extension metadata via two objects:

Object	Description
EbiDocExtnMeta	A holder for all the extension metadata for all the fields
EbiDocExtnMetaInfo	Associates a field with a set of values

After you create an EbiDocExtnMetaInfo object for a specific field, you set the values for the field as an array, even if there is only one value. The type of the array must correspond to the data type of the field.

After you've created an EbiDocExtnMetaInfo object for each field and added it to the EbiDocExtnMeta object, call setExtensionMetaData() for EbiAddDocumentParams to associate it with the document you are adding.

Getting fields for the document type

To find out what fields to specify for a document, you can get a collection of EbiDocField objects for the document type. This example presents a method called **getDocTypeFields()** that gets all the document type fields to which the user has READ access.

The getDocTypeFields() method needs to access a content manager (EbiContentMgmtDelegate) and context object (EbiContext), which are passed in as arguments:

```
public void getDocTypeFields(EbiContentMgmtDelegate cmgr, EbiContext context)
        throws EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException
```

```
    {
        EbiDocType docType = cmgr.getDocumentTypeByName(context, "Movie Review");
        if (docType != null)
        {
            Collection fields = cmgr.getFilteredDocumentFields(context,
docType.getDocTypeID());
            System.out.println("Fields: " + fields);
        }
    }
```

Getting a field object by name

You can also get individual fields by name. This example presents a method called **getField()** that gets the field named **Director**.

The getField() method needs to access a content manager (EbiContentMgmtDelegate) and context object (EbiContext), which are passed in as arguments:

```
public void getField(EbiContentMgmtDelegate cmgr, EbiContext context)
        throws EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException
    {
        EbiDocField fldDirector = cmgr.getDocumentFieldByName(context, "Director");
        System.out.println("Director field: " + fldDirector);
    }
```

Setting a field value

This example presents a method called **setFieldValues()** that performs the following tasks:

- Creates an EbiDocExtnMeta holder and EbiDocExtnMetaInfo objects for the Director and Genre fields
- Associates the EbiDocExtnMeta object with an EbiAddDocumentParams object that it uses to add the new document to the content respository

The values for each field are passed as String arrays.

Note that the setFieldValues() method does not set the ACL for the new document. This means that the ACL is null and the document inherits the ACL of its folder.

The setFieldValues() method needs to access a content manager (EbiContentMgmtDelegate) and context object (EbiContext), which are passed in as arguments:

```
public void setFieldValues(EbiContentMgmtDelegate cmgr, EbiContext context)
        throws EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException
    {
        // Get the doctype
        EbiDocType type = cmgr.getDocumentTypeByName(context, "Movie Review");
```

```
    // Get the folder
    EbiDocFolder folder = (EbiDocFolder)cmgr.lookupDirectoryEntry(
        context, "MyApp/MovieReviews/Current", EbiDocFolder.EL_DOC_FOLDER);
    // Instantiate a document addition parameters object
    EbiAddDocumentParams docParams = cmgr.createAddDocumentParams();

    // Create the extension metadata holder object
    EbiDocExtnMeta meta = cmgr.createExtnMeta();

    // Specify the extn metadata field values for 'Director'
    EbiDocField fldDirector = cmgr.getDocumentFieldByName(context, "Director");
    EbiDocExtnMetaInfo miDirector = cmgr.createExtnMetaInfo(fldDirector);
    String[] directors = { "Andy Wachowski", "Larry Wachowski" };
    miDirector.setFieldValues(directors);
    meta.setExtnMetaInfo(miDirector);

    // Specify the exnt metadata field values for 'Genre'
    EbiDocField fldGenre = cmgr.getDocumentFieldByName(context, "Genre");
    EbiDocExtnMetaInfo miGenre = cmgr.createExtnMetaInfo(fldGenre);
    String[] genres = { "Action", "Thriller", "Sci-Fi" };
    miGenre.setFieldValues(genres);
    meta.setExtnMetaInfo(miGenre);

    // Get the content
    String movieContent = "This movie has exceeded all expectations!....";
    byte content[] = movieContent.getBytes();

    // Set the extension metadata into the doc params object
    docParams.setExtensionMetaData(meta);

    docParams.setName("The Matrix (1999)");
    docParams.setDocTypeID(type.getDocTypeID());
    docParams.setFolderID(folder.getID());
    docParams.setAuthor("Night Ghost");
    docParams.setTitle("The Matrix (1999)");
    docParams.setMimeType("text/xml");
    docParams.setContent(content);
    docParams.setComment("Initial revision.");

    // params.setAcl(...); specify an ACL, otherwise inherit ACL of parent folder

    EbiDocument doc = cmgr.addDocument(context, docParams);

    // Publish the new document
    cmgr.publishDocumentContentVersion(context, doc.getID(), 1, true, true);
}
```

Getting all fields

This example presents a method called **getExtnMeta()** that gets all the extension metadata fields for a specified document. The method uses the EbiDocExtnMeta object as a holder for the document's fields. This object provides methods for getting information about the fields, such as names and values.

The getExtnMeta() method needs to access a content manager (EbiContentMgmtDelegate), context object (EbiContext), and the document of interest—all of which are passed in as arguments:

```
public void getExtnMeta(EbiContentMgmtDelegate cmgr, EbiContext context, String docID)
        throws EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException
    {
        // Get the extension metadata holder for the document
        EbiDocExtnMeta extnMetaData = cmgr.getDocumentExtnMeta(context, docID);
        System.out.println("Extension metadata: " + extnMetaData);

        // Enumerate the field names
        Iterator fieldNames = extnMetaData.getFieldNames().iterator();
        while (fieldNames.hasNext())
            System.out.println("Field: " + (String)fieldNames.next());

        // For each extension meta info
        for (int i = 0; i < extnMetaData.size(); i++)
        {
            EbiDocExtnMetaInfo mi = extnMetaData.getExtnMetaInfoByIndex(i);
            System.out.println("MetaInfo " + i + ": " + mi);

            String fieldName = mi.getFieldName();
            System.out.println("Field name: " + fieldName);

            Collection fieldValues = mi.getFieldValues(false);
            System.out.println("Values: " + fieldValues);
        }
    }
```

Getting field values for a single field

This example presents a method called **getExtnMeta()** that gets an EbiDocExtnMetaInfo object for a single field.

The getExtnMeta() method needs to access a content manager (EbiContentMgmtDelegate), context object (EbiContext), and the document and field of interest—all of which are passed in as arguments:

```
public void getExtnMeta(
        EbiContentMgmtDelegate cmgr, EbiContext context,
        String docID, String fieldID)
        throws EboUnrecoverableSystemException,
EboSecurityException, EboItemExistenceException
```

```
{
    EbiDocExtnMetaInfo info = cmgr.getDocumentExtnMetaInfo(
        context, docID, fieldID);
    System.out.println("Meta Info: " + info);
}
```

From the EbiDocExtnMetaInfo object you can get a Collection of the values for the field (the values of the array that set the field). A boolean argument lets you specify whether the data type of the returned values is String or the actual data type of the field.

This statement gets the values of the EbiDocExtnMetaInfo object as Strings:

```
Collection valueStrings = info.getFieldValues(true);
```

Methods for managing documents

This table lists methods that let you manage documents, edit the metadata, and get documents:

Method	Returns	Description
createAddDocument Params()	EbiAdd-DocumentParams	Creates an empty object to hold the data needed to add a document, including metadata, extension metadata, content, and ACL. You use the EbiAddDocumentParams object with the addDocument() method.
addDocument()	EbiDocument	Adds a document to the content repository. Content for the document is optional.
copyDocument()	EbiDocument	Copies a document to a folder or to a parent document.
getDocument()	EbiDocument	Gets a document object for a specified document ID.
moveDocument()	EbiDocument	Moves a document to a folder or to a parent document.

Method	Returns	Description
updateDocument()	void	Updates the information about a document in the content repository using changes made to EbiDocument.
removeDocument()	boolean	Removes a document and all its versions from the system.
addDocumentCategory-Reference()	void	Adds a document to a category. A document can belong to many categories.
removeDocument Category-Reference()	boolean	Removes a document from a category.
getDocumentExtnMeta()	EbiDocExtnMeta	Gets a holder for the extension metadata objects associated with each field of the document. Its methods let you get the values for individual fields.
getDocumentExtnMeta-Info()	EbiDocExtnMeta Info	Gets the extension metadata object for a field of the document.
getDocumentsByType() and getFilteredDocuments-ByType()	Collection of EbiDocument	Gets a collection of the documents of a particular document type. The filtered version omits documents to which the current user has no READ access. The unfiltered version gets all documents for the type, regardless of access rights.
getLatestDocumentContent-Version()	EbiDocVersion	Gets the most recent version of a document.
getDocumentContent-Version()	EbiDocVersion	Gets a version of a document.
getDocumentContent-Versions()	Collection of EbiDocContent	Gets all the versions of a document.
publishDocumentContent-Version()	void	Publishes a version of a document.

Method	Returns	Description
getContent()	EbiDocContent	Gets the published content for a document. If the document is not published, returns null.
unpublishDocumentContent()	boolean	Removes a document's content from the published area. The content for all document versions remains intact.

Specifying layout sets for documents

Typically, the layout styles associated with the document type are adequate for displaying your document (as described in "Managing layout styles" on page 38). When you have hundreds of documents (news stories, press releases, editorials, reviews), you don't want to design custom XSL for each one. One design or a few alternative designs are enough; you can associate one or more layout styles with a document type.

When to use a layout set

When you want to lock in a particular layout for an individual document, you can specify a *layout set* for that document. A layout set uses a specific layout style, selected from the ones that are valid for the document's type. The layout set uses one or more of the layout document descriptors associated with that style. In the set you can use whatever version of the layout document is currently published or you can select a specific version. The set needs to includes layout document descriptors for whatever clients will view the content. The XSL in the layout documents associated with the descriptors render the document.

What a layout set is good for A layout set is meant for locking in a presentation so that the document always looks the same. As layout styles for a document type evolve with new versions, the presentation of an individual document will change. Use a layout set when it is important to preserve the original presentation.

What a layout set is less appropriate for The layout set is less appropriate for giving a document a unique look. It may be more appropriate to add a new document type. However, you can also add a custom style to the document type in order to make a special layout available for the document. If you don't want to be constrained to styles for the document type, you could design your application to locate style documents another way—for example, via a custom field. However, you would want to make sure your custom system has the flexibility for getting different XSL for different clients.

Methods for managing layout sets

These methods of EbiContentMgmtDelegate manage layout sets:

Method	Returns	Description
createDocLayoutSet()	EbiDocLayoutSet	Creates an empty layout set. It is associated with a document when you call addDocument().
getDocumentLayoutSet()	EbiDocLayoutSet	Gets the layout set for a document.
removeDocumentLayoutSet()	boolean	Removes the layout set from the document.
updateDocumentLayoutSet()	void	Updates the layout set with new layout style and layout style descriptor information.

To associate a layout set with a new document, call the setLayoutSet() method of EbiAddDocumentParams.

To change the XSL documents in the layout set of an existing document, call getDocumentLayoutSet(), call methods of EbiDocLayoutSet to make changes, and then call updateDocumentLayoutSet().

NOTE: Currently, you cannot add a layout set to a document if it didn't have one when it was added.

Creating links between documents

You can specify relationships between documents by specifying that one document is a child of another.

This section includes these topics:

- Two types of document relationships
- Hierarchical linking
- Adding a child document
- Compound linking
- Linking a child document
- Updating a link with a new document version
- Getting linked parent documents
- Getting linked child documents

Two types of document relationships

The content repository supports two types of document relationships—*hierarchical* and *compound*

Document relationship	Description
Hierarchical	Where each document in the hierarchy stores the ID of its parent document. A document has only one parent. The value –1 identifies the top document in a chain of links. The chain can be an indefinite number of levels deep. Hierarchical linking is designed for a threaded discussion and similar structures.
Compound	Where a link object identifies the originator of the link (parent) and the target of the link (child). A parent can have many child documents, and a child can have many parents. Compound linking is designed for building composite documents, where many pieces of content are brought together in a single presentation page. For example, child documents might include sections of a report, a list of cross-references that is appended to a document, or images to be displayed in a page.

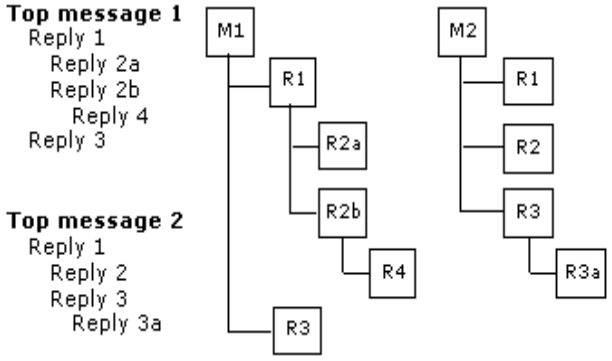
You can use linked documents in many ways. A parent document might serve as a container of child documents, where each subsection of the document is produced by a different author. Documents could be linked in a chain to identify a message thread. Links could point to nontext documents that are stored separately, such as images or sound files.

CAUTION: *When specifying either hierarchical or compound links, you are not prevented from creating circular links, where a parent document is also a child of its child document. If you do this, proceed with caution: circularity may confuse both programmer and end user. It is up to you to understand the link structure of your repository when you process the content.*

Hierarchical linking

Hierarchical linking lets you create a threaded discussion. The following diagram shows two views of a threaded discussion. Each reply has one parent, and each message can be the parent of several replies. The top message in each chain has no parent.

When a user submits a reply, the application uses the ID of the original message as the parent of the new reply document:



Methods for hierarchical linking

These methods of EbiContentMgmtDelegate are useful in managing hierarchical links:

Method	Returns	Description
addDocument()	EbiDocument	When adding a document, you can make it a child document by specifying a parent ID. If the parent ID is –1, the document has no parent.
getChildDocuments() and getFilteredChild-Documents()	Collection of EbiDocument	Gets child documents that have a parent ID of the specified document. The filtered version omits documents to which the current user has no READ access. The unfiltered version gets all child documents, regardless of access rights.

In addition, when you have an EbiDocument object, you can get and change the parent document ID, via getParentDocID() and setParentDocID(). After changing the ID, call updateDocument() to put the changes in the repository.

Adding a child document

This example presents a method called **addChildDocument()** that creates a child document as a reply in a message thread. Inside a **while** loop, the method navigates the thread to the top message, and then uses its title to construct the name and subtitle of the reply.

The addChildDocument() method needs to access a content manager (EbiContentMgmtDelegate), the context object (EbiContext), the parent document, a message subject, and a reply—all of which are passed in as arguments:

```
public void addChildDocument(
        EbiContentMgmtDelegate cmgr, EbiContext context, String
folderID,
        String parentID, String subject, String reply)
        throws EboUnrecoverableSystemException,
EboSecurityException, EboItemExistenceException
    {
        EbiAddDocumentParams params =
cmgr.createAddDocumentParams();
        params.setName("Reply to " + threadTitle);
        EbiDocType doctype = cmgr.getDocumentTypeByName(context,
"Discussion");
        if (doctype != null)
            params.setDocTypeID(doctype.getDocTypeID());
        params.setFolderID(folderID);
        params.setAuthor(context.getUserID());
        params.setTitle(subject);
```

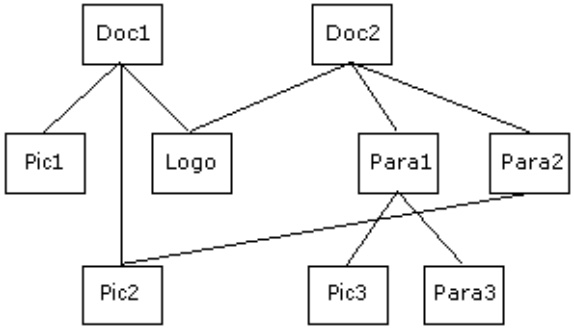
```
            params.setSubtitle(threadTitle);
            params.setMimeType("text/plain");
            params.setContent(reply.getBytes());
            params.setParentID(parentID);
            cmgr.addDocument(context, params);
    }
```

Compound linking

Compound linking lets you create a network of interrelated documents. You might use it to create a composite document out of many contributed pieces, such as sections (written by different authors), images, cross-references, and other information.

For more information, see "Composite documents" on page 74.

The following diagram shows a network of documents that are used by two different parent documents; some of the material is shared by both:



Access to documents you want to link To create a link, you must check out both the parent and child documents, add the link, and then check in both documents.

XML for composite documents It is easy to program the display of a composite document when the content type is XML. Your portlet inserts each child document as a node in the DOM with an appropriate element name. An XSL style sheet specifies how those elements are displayed. You don't have to insert the child documents into existing content in any particular order. The order is determined by the style sheet. By selecting different style sheets, you can change the way the different elements are displayed and whether they are included at all.

For information about style sheets for document types, see "Managing layout styles" on page 38. To specify styles for individual documents, see "Specifying layout sets for documents" on page 60.

For example, suppose you have a Movie Review document type and the paragraphs of the review are its content. Child documents for the review could include an image from the movie and biographies of the cast. In the displayed HTML, the biographies could be displayed on the same page or they could be links to another HTML page. You could have different style sheets that determine which way to display the biographies and whether the image is on the left or the right.

Methods for compound linking

NOTE: When adding, removing, and changing links, you must check out the parent and child documents.

These methods of EbiContentMgmtDelegate are useful in managing compound links:

Method	Returns	Description
addDocumentLink()	EbiDocLink	Adds a link between two documents. For the child version ID argument, you can specify a specific version or –1 to use the published version.
getDocumentLink()	EbiDocLink	Gets a link object, given the parent and child IDs.
removeDocumentLink()	boolean	Removes a link.
updateDocumentLink()	void	Allows you to change the version of the child document the link uses.
getLinkChildDocuments() and getFilteredLinkChild-Documents()	Collection of EbiDocument	Gets the link objects for the child documents that are linked to the specified parent document. The filtered version omits documents to which the current user has no READ access. The unfiltered version gets all documents regardless of access rights.
getLinkParentDocuments() and getFilteredLinkParent-Documents()	Collection of EbiDocument	Gets the document objects for the parent documents to which the specified child document is linked. The filtered version omits documents to which the current user has no READ access. The unfiltered version gets all documents regardless of access rights.

Linking a child document

This example presents a method called **addDocLink()** that adds a link between a parent document and a child document.

The addDocLink() method needs to access a content manager (EbiContentMgmtDelegate), the context object (EbiContext), the parent and child documents, and the child document version—all of which are passed in as arguments.

```
public void addDocLink(
        EbiContentMgmtDelegate cmgr, EbiContext context,
        String linkParentDocID, String linkChildDocID, int
linkChildVersionID)
        throws EboUnrecoverableSystemException,
EboSecurityException, EboItemExistenceException
    {
        EbiDocLink lnk = cmgr.addDocumentLink(
            context, linkParentDocID, linkChildDocID,
linkChildVersionID);
    }
```

Updating a link with a new document version

This example presents a method called **updateDocumentContentAndLink()** that creates and publishes a new version of a child document, then updates the link from the parent to point to the new version.

If a new version of the child document is published later, this link continues to point to the old version. A link between parent and child must exist. If not, you need to use addDocumentLink() instead of updateDocumentLink().

The updateDocumentContentAndLink() method needs to access a content manager (EbiContentMgmtDelegate), the context object (EbiContext), the parent and child documents, document content, and a MIME type—all of which are passed in as arguments:

```
public void updateDocumentContentAndLink(
        EbiContentMgmtDelegate cmgr, EbiContext context,
        String linkParentDocID, String linkChildDocID,
        byte[] linkChildDocContent, String linkChildDocMimeType)
        throws EboUnrecoverableSystemException,
EboSecurityException, EboItemExistenceException
    {
        // Create a new version of the link child document
        int newVersionID = cmgr.checkinDocument(
            context,                    // Context
            linkChildDocID,             // Docid of link child
            linkChildDocMimeType,       // Mime type
            linkChildDocContent,        // New content
            "new version",              // Check-in comment
            false);                     // Whether to keep doc
                                       // checked out
```

```
        // Publish it
        cmgr.publishDocumentContentVersion(
            context, linkChildDocID, newVersionID, true, true);

        // Now update the link to point to the new version
        cmgr.updateDocumentLink(
            context,                    // Context
            linkParentDocID,            // Link parent docid
            linkChildDocID,             // Link child docid
            newVersionID);              // New version id
    }
```

Getting linked parent documents

This example presents a method called **getLinkParentDocuments()** that gets the parent documents that are linked to a specified child.

By calling getFilteredLinkParentDocuments(), the code retrieves only documents to which the user has READ access.

The getLinkParentDocuments() method needs to access a content manager (EbiContentMgmtDelegate), the context object (EbiContext), and the child document of interest—all of which are passed in as arguments:

```
public void getLinkParentDocuments(
        EbiContentMgmtDelegate cmgr, EbiContext context,
        String linkChildDocID)
        throws EboUnrecoverableSystemException,
EboSecurityException, EboItemExistenceException
    {
        Collection linkParentDocs =
cmgr.getFilteredLinkParentDocuments(context, linkChildDocID);
        System.out.println("Parent docs: " + linkParentDocs);
    }
```

Getting linked child documents

This example presents a method called **getLinkChildDocuments()** that gets the child documents that are linked to a specified parent.

By calling getFilteredLinkChildDocuments(), the code retrieves only documents to which the user has READ access.

The getLinkChildDocuments() method needs to access a content manager (EbiContentMgmtDelegate), the context object (EbiContext), and the parent document of interest—all of which are passed in as arguments.

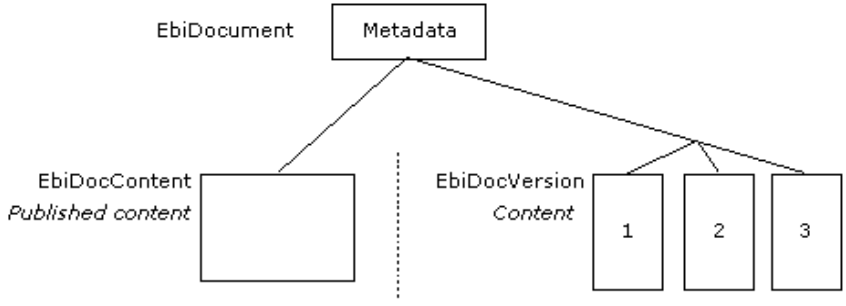
```
public void getLinkChildDocuments(
        EbiContentMgmtDelegate cmgr, EbiContext context,
        String linkParentDocID)
        throws EboUnrecoverableSystemException,
EboSecurityException, EboItemExistenceException
    {
        Collection linkChildDocs =
cmgr.getFilteredLinkChildDocuments(context, linkParentDocID);
        System.out.println("Child docs: " + linkChildDocs);
    }
```

Modifying and publishing documents

The CM subsystem includes functionality that supports checkout, checkin, versioning, and publishing.

Much of the information in the CM subsystem is data about documents. However, when you start using the checkout and checkin methods, you also get multiple versions of document content. Each time a document is checked in, a new version is created. When a document is published, there is also a released version of the content, which comes from the set of versions. You can continue creating new versions, while the publicly available version remains stable.

The diagram that follows shows the relationship between an EbiDocument object, which holds the document metadata, and its version objects. The content for each version is stored in an EbiDocVersion object. When you select a version for publishing, that version's content is copied to an EbiDocContent object.



NOTE: It is important to remember that only the content has multiple versions. There is only one version of the document's metadata.

You might program portlets for source control and publishing to accomplish tasks like these:

Task	Information
Add a new document to the system	If the document is added with accompanying content, the system creates a first version.
Check out a document	When a user checks out the document, your portlet copies the content to an appropriate editing environment.
Check in a document	When the user checks in the document, the system creates a new version.
Publish documents	You might have a scheduled task that checks publish dates and calls publishDocumentContentVersion() when a document's publish date is passed.
Unpublish documents	You might have a scheduled task that removes a published version when the expiration date has passed. The task might move the document to an archive folder, purge it from the system, or set its publish date so another version can be published later.
Review the checkin comments for a document's versions.	—

Tracking document status

To find out if a document is published, you call the EbiDocument method getPublishStatus(). If it returns null, then the document has no published content.

A publish date does not automatically reflect the time the document was published. It just indicates when it should be published; for example, a publish date of null means publish immediately. However, your publishing portlet can set the publish date if you want to track the date a document became available.

The document's status field is available for your own document tracking. You can establish your own application-specific set of status values and update the document's status field to reflect its progress through your document processing procedures. For example, you could specify submitted, reviewed, approved, rejected, published, unpublished, archived, and purged as status values for your application.

Setting document status This example presents a method called **setDocumentStatusToRejected()** that sets a document's status to **rejected**—perhaps to indicate that the document has been rejected by a content administrator and requires further changes before it can be published. Note that after setting status, you must call the updateDocument() method for the change to take effect.

The setDocumentStatusToRejected() method needs to access a content manager (EbiContentMgmtDelegate), the context object (EbiContext), and the document of interest—all of which are passed in as arguments:

```
public void setDocumentStatusToRejected(
        EbiContentMgmtDelegate cmgr, EbiContext context, String
docID)
        throws EboUnrecoverableSystemException,
EboSecurityException, EboItemExistenceException
    {
        EbiDocument doc = cmgr.getDocument(context, docID);
        doc.setStatus("rejected");
        cmgr.updateDocument(context, doc);
    }
```

Methods for source control and publishing

These methods of EbiContentMgmtDelegate are available for source control and publishing:

Method	Returns	Description
checkoutDocument()	boolean	Checks out a document to the current user (specified in the context argument). This method locks the document. To get the content for editing, use other methods—such as getDocumentContentVersion().
checkinDocument()	int	Checks in a new version of the document with data for the content. Only the user who checked out the document can check it in. The user is implicit in the context argument.
uncheckoutDocument()	boolean	Releases the lock set by the current user.
unlockDocument()	boolean	An administrative method that allows you to release a document lock that was set by any user.

Method	Returns	Description
rollbackDocument-Content()	void	Rolls document content from the latest version back to the specified one.
publishDocument-ContentVersion()	void	Publishes a specific version of the specified document.
getContent()	EbiDocContent	Gets the published content object for a document. You can choose whether it includes the actual data. If it does, get the byte array of data by calling getData() of EbiDocContent.
getDocumentContent-Version()	EbiDocVersion	Gets a version of a document. You can choose whether it includes the actual data. If it does, get the byte array of data by calling getData() of EbiDocVersion.
unpublishDocument-Content()	boolean	Removes the published content for a document.

Displaying documents

Portlets in your online application get a document's metadata and content, retrieve linked content, and use the associated layout styles to display the document to the user.

HTML content

If the content type of a document is HTML and it has no linked documents, the portlet might simply get and set the content, as shown below:

```
EbiContentMgmtDelegate cm = null;
try {
  cm =
   com.sssw.cm.client.EboFactory.getDefaultContentMgmtDelegate();
} catch (EboFactoryException ebfe) {
  throw new EboUnrecoverableSystemException(ebfe,
  "Unable to get ContentManager");
}
try {
  EbiDocument doc = (EbiDocument)
   cm.lookupDirectoryEntry(context, "MyFolder/TDBDoc1",
   EbiDocument.EL_DOCUMENT);
   EbiDocContent content = cm.getContent(context, doc.getID(),
true);
```

```
            if (content != null)
            {
              PrintWriter writer = response.getWriter();
              byte [] html = content.getData();
              String shtml = new String(html);
              writer.print(shtml);
            }

          } catch (EboItemExistenceException eiee)
          {
            throw new EboUnrecoverableSystemException(eiee,
             "Unable to get Content");
          } catch (EboSecurityException ese)
          {
            throw new EboUnrecoverableSystemException(ese, "Security
          exception");
          }
```

XML content

If the content of a document is an XML String and it has no linked documents, the portlet could get the content and the document layout (also as an XML String) and use the layout XSL to transform the XML.

This concept is illustrated in the **displayContent()** example method shown below. In this example, methods in com.sssw.fw.util.EboXmlHelper convert a String to a DOM and apply an XSL transformation to a DOM. The displayContent() method accesses a content manager (EbiContentMgmtDelegate), the context object (EbiContext), and the document of interest—all of which are passed as arguments:

```
public void displayContent(
        EbiContentMgmtDelegate cmgr, EbiPortalContext context, String docID)
        throws EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException
    {
        EbiDocument doc = (EbiDocument)cmgr.lookupDirectoryEntry(
            context, "MyFolder/TDBDoc1", EbiDocument.EL_DOCUMENT);
        EbiDocContent doccnt = cmgr.getContent(context, doc.getID(), true);
        if (doccnt != null)
        {
            byte[] xml = doccnt.getData();
            EbiDocVersionDescriptor layoutver = cmgr.getDocumentLayout(
                context, docID, EbiContentMgmtDelegate.COMPARE_ALL, true);
            EbiDocVersion layoutcnt = cmgr.getDocumentContentVersion(
                context, layoutver.getDocumentID(), layoutver.getDocumentVersionID(),
true);
            byte[] xsl = layoutcnt.getData();
            String sxml = new String(xml);
            String sxsl = new String(xsl);

            String content = EboXmlHelper.processXML(
                EboXmlHelper.getDOM(sxml), EboXmlHelper.getDOM(sxsl));
```

```
            // Set type according to results of xsl transformation
            response.setContentType(EbiPortletConstants.MIME_TYPE_HTML);
            // Use a PrintWriter to render
            writer.print(content);
        }
    }
```

Composite documents

A composite document could be constructed in many different ways. It is up to your portlet to gather the pieces and put them together in an appropriate way. Typically, you would build an XML DOM for the composite document and add elements for each piece. For a simpler composite document where the pieces are HTML fragments, you might concatenate them into a larger HTML fragment.

To illustrate the process of building an XML DOM, suppose you are displaying a movie review, a document of type Movie Review. The content of the movie review document is the text paragraphs of the review. The document's metadata provides the title, author, and other information specific to the Movie Review type, such as genre, director, year of release, and cast. Child documents refer to an image of the movie and cast biographies. To display all the data, the portlet builds an XML DOM of the pieces and provides an XSL style sheet for display specifications.

You will want to plan an XML structure for defining the XSL and building the DOM in the portlet's code. You may want to formalize that structure in a DTD. The XML structure might look like this (shown without closing tags):

```
<REVIEW>
   <TITLE>
   <AUTHOR>
   <GENRE>
   <DIRECTOR>
   <CAST>
      <CASTMEMBER>
         <CASTPICTURE>
         <BIO>
      </CASTMEMBER>
      <CASTMEMBER>
         <CASTPICTURE>
         <BIO>
      </CASTMEMBER>
   </CAST>
   <CONTENT>
</REVIEW>
```

The coding steps might be:

1. Get the EbiDocument object.
2. Get the metadata you want displayed (such as title, author, director, and genre) and add elements for each one. Element names might be TITLE, AUTHOR, and so on. The data values could be attributes or text nodes of the elements.
3. Get the cast metadata and add a CAST element, with child CASTMEMBER elements for each one.
4. Get the content data. Add a CONTENT element for the review paragraphs (the document content) and add the content data as a text node of the element.
5. Call getLinkChildDocuments() to get the linked child documents.
6. For each linked document, get the MIME type and other information to determine the document's purpose:
   - For an image from the film, add a MOVIEPICTURE element whose attributes have information needed by the XSL to build an image link.
   - For a cast biography, find the corresponding CASTMEMBER element and add a child BIO element. Depending on the page design, you could insert information to build a link or include the paragraphs.
   - For a picture of a cast member, find the CASTMEMBER element and add a CASTPICTURE element with information to build an image link.
7. When the XML DOM is complete, call methods of the context object to set the MIME type and the content.

4 Securing Content

This chapter describes how to use ACL-based security to authorize access to Content Management (CM) subsystem elements. It has these sections:

- About access control
- ACL-based security
- Methods for managing access control
- Examples of adding ACLs
- Example of handling a security exception

NOTE: Most of the security tasks described in this chapter can also be accomplished using the CMS Administration Console.

For more information, see Chapter 19, “Managing Content Security”.

About access control

The CM subsystem supports ACL-based security, as described in “ACL-based security” on page 78. You can specify access restrictions based on user ID or group membership on most objects in the CM subsystem. You can use access restrictions to:

- Prevent changes after your infrastructure of document types, folders, and categories has been set up
- Prevent inadvertent deletion of objects
- Protect documents or other objects from being seen by unauthorized users

CM user groups

A comprehensive security policy must set different permissions for different user roles. Typical roles in the CM subsystem are:

Role	Description
Author	Has read and write access for documents; has read, write, and list access for folders and categories.
Publisher	Has publish access for documents; has list access for folders and categories.
Administrator	Has all access rights to all objects. Users are considered administrators when the ACL assigned to the EbiContentAdmin interface gives them at least one of the permissions. See "ContentAdmin group" on page 80.

When setting up users and groups for exteNd Director, you will want to consider how your users fall into these roles and create appropriate groups. You can use those user IDs and groups to create ACLs that implement your security. You might create a master ACL that you can get and reuse throughout the CM subsystem.

You can set up users and groups using the Director Administration Console (DAC).

 For more information, see the chapter on using the Directory section of the DAC in the *User Management Guide*.

ACL-based security

You specify access restrictions on CM objects by using an access control list (ACL). To provide support for ACLs, exteNd Director implements the java.security.acl.Acl interface. Each of the securable *elements* has a set of supported access right types, or *permissions*. The supported permissions are defined as String constants in each object's interface.

This section describes using ACLs to specify access restrictions on CM objects.

 For general information about using ACLs in exteNd Director applications, see the chapter on ACL-based security in the *User Management Guide*

Permissions

The permissions defined for the CM subsystem include:

Permission	Description
PROTECT	Allows the users and groups in the ACL to change permissions on the object.
READ	Allows the users and groups in the ACL to view the object or get the metadata for an object.
WRITE	Allows the users and groups in the ACL to make changes to the object, by updating the object programmatically or by checking in a new version of a document. A user who has been denied WRITE access cannot check out a document.
LIST	Allows the users and groups in the ACL to view a list of the objects that this object contains. This includes the documents and subfolders of a folder, the documents and subcategories of a category, and the documents associated with a document type.
PUBLISH	For documents, allows the users and groups in the ACL to change the published status of the document. They can publish it and remove it from the published area.

Element types and associated permissions

The table that follows lists the subsystem securable element types (not including some securable superinterfaces) and permissions they support:

Object	Access right types
EbiContentAdmin	PROTECT, READ, WRITE
EbiDocType	PROTECT, READ, WRITE, LIST
EbiDocField	PROTECT, READ, WRITE
EbiDocCategory	PROTECT, READ, WRITE, LIST
EbiDocFolder	PROTECT, READ, WRITE, LIST
EbiDocument	PROTECT, READ, WRITE, PUBLISH
EbiDocLayoutStyle	PROTECT, READ, WRITE
EbiLayoutDocDescriptor	PROTECT, READ, WRITE

ContentAdmin group

The EbiContentAdmin interface represents the built-in content administrator group. Users added to this group have specified access to subsystem management and administration. Here are the available permissions:

Permission	Description
PROTECT	Set ACLs for the ContentAdmin type.
READ	Get subsystem elements (folders, categories and documents) in the CM subsystem
WRITE	Add subsystem elements to the CM subsystem

Methods for managing access control

The EbiContentMgmtDelegate interface provides access to most of the security-related methods in the CM subsystem.

Accessing ACLs for existing elements

These methods of EbiContentMgmtDelegate let you set security for objects:

Method	Returns	Description
getAcl()	java.security.acl.-Acl	Gets the ACL for a securable element: category, field, folder, layout style, layout document descriptor, document type, or document.
setAcl()	void	Assigns an ACL to a securable element.
removeAcl()	boolean for success	Removes the ACL currently set for an element.
isAuthorized()	boolean	Checks whether the user identified in the context object is authorized for the specified type of access for an object.
getAllAccessible()	Collection	From a list of securable elements, filters out the ones that are accessible to the user whose context is passed in.

Method	Returns	Description
getAdminElement()	EbiContentAdmin	Gets the Content Admin element holding the ACL that identifies the users and groups that have administrator access to content objects. See "Accessing ACLs for ContentAdmin" on page 82.
hasAdminAccess()	boolean	This is a shortcut for isAuthorized() and is invoked with a reference to the Content Admin element.

Specifying ACLs for new elements

For securable elements, you can specify an ACL when you create the object. It is an argument of the object's **add** method on the EbiContentMgmtDelegate — addDocument(), addFolder(), and so on.

For a code example, see "Examples of adding ACLs" on page 82.

Inheriting ACLs

For the following objects: if you don't specify an ACL when you create them, the settings of their containers are copied to the new object:

New	Copy the ACL of their
Folders	Parent folder
Documents	Folder
Layout descriptors	Layout style

After the object is created, there is no further connection to the container's ACL. Changes to a container's ACL have no effect on the contained objects.

For other object types: if you don't specify an ACL, they have an empty ACL.

Accessing ACLs for ContentAdmin

These methods on EbiContentAdmin allow you to access ACLs for the ContentAdmin group:

Method	Returns	Description
getAcl()	java.security.acl.-Acl	Gets the ACL currently set for the ContentAdmin element.
setAcl()	void	Assigns an ACL to the Content Admin element.
removeAcl()	boolean for success	Removes the ACL currently set for the Content Admin element.
isUserAuthorized()	boolean	Checks if the current user is listed in the Content Admin ACL.

Restricting element access to administrators

You can restrict access for any CM element to Content Admin users using the setRestrictedAccess() method. Specify the permission you want to restrict. For example, if you restrict access to a folder for the WRITE permission, only members of the ContentAdmin group have WRITE access to the element.

NOTE: The restricted access right takes precedence over any other ACL associated with the restricted element.

Here are the related methods on the EbiSecurityManager interface:

Method	Returns	Description
setRestrictedAccess()	boolean for success	Restricts specified access for an element to system administrators
check RestrictedAccess()	boolean	Checks whether an element has restricted access

Examples of adding ACLs

This example presents a method called **demonstrateSecurity()** that illustrates the following techniques:

- Adding READ access to the ContentAdmin element associated with a *principal* (the identity assigned to a user as a result of authentication)
- Adding a folder with an ACL

- Adding a folder with no ACL

  NOTE: In this case the folder inherits the ACL from its parent.

- Adding an ACL to an existing folder

The demonstrateSecurity() method needs to access a content manager (EbiContentMgmtDelegate), the context object (EbiContext), and a principal—all of which are passed in as arguments:

```
public void demonstrateSecurity(
        EbiContentMgmtDelegate cmgr, EbiContext context,
        Principal principal)
        throws
            EboUnrecoverableSystemException, EboSecurityException,
EboItemExistenceException, EboFactoryException, NotOwnerException
    {
        EboPermission readPerm = EboPermission.getPermission(
            context.getEbiSession(), EboPermission.READ);
        EboPermission writePerm = EboPermission.getPermission(
            context.getEbiSession(), EboPermission.WRITE);

        // Add READ access to the Content Admin element to the passed-in principal
        EbiContentAdmin adminElement = cmgr.getAdminElement(context);
        Acl admAcl = cmgr.getAcl(context, adminElement);
        AclEntry aclEntry = com.sssw.fw.factory.EboFactory.getAclEntry();
        aclEntry.setPrincipal(principal);
        aclEntry.addPermission(readPerm);
        admAcl.addEntry(principal, aclEntry);
        cmgr.setAcl(context, adminElement, admAcl);

        // Add a folder with an ACL
        Acl acl = com.sssw.fw.factory.EboFactory.getAcl();
        aclEntry = com.sssw.fw.factory.EboFactory.getAclEntry();
        aclEntry.setPrincipal(principal);
        aclEntry.addPermission(readPerm);
        aclEntry.addPermission(writePerm);
        cmgr.addFolder(
            context,
            cmgr.getRootFolder(context),
            "Movie Reviews",
            EbiDocFolder.DIR_TYPE_DEFAULT,
            "Folder for movie reviews",
            acl);

        // Add a folder with no ACL -- it will inherit the ACL
        // from its parent folder (if there is an ACL set on the parent)
        EbiDocFolder frFolder = cmgr.addFolder(
            context,
            cmgr.getRootFolder(context),
            "Financial Reports",
            EbiDocFolder.DIR_TYPE_DEFAULT,
            "Folder for financial reports",
            null);
```

```
        // This code adds an ACL to an existing folder.
        cmgr.setAcl(context, frFolder, acl);
    }
```

Example of handling a security exception

This example presents a method called **demonstrateHandleExceptions()** that illustrates how to handle a security exception (and other exceptions as well).

This code publishes version 2 of a document whose ID is assigned to the variable **docid**. The **publishDocumentContentVersion()** method will throw an EboSecurityException if the user is not allowed to publish the specified document. This example handles the exception by adding an error message to the context object. The portlet can then include the error message in its generated content so the user knows what went wrong.

The demonstrateHandleExceptions() method needs to access a content manager (EbiContentMgmtDelegate), the context object (EbiContext), and the document of interest—all of which are passed in as arguments.

```
public void demonstrateHandleExceptions(
        EbiContentMgmtDelegate cmgr, EbiContext context, String docID)
    {
        try
        {
            cmgr.publishDocumentContentVersion(context, docID, 2, true, true);
        }
        catch (EboSecurityException se)
        {
            se.printStackTrace();
            String msg = "Security violation: " + se.toString();
            context.setValue("error", "User does not have access. " + msg);
        }
        catch (EboUnrecoverableSystemException use)
        {
            use.printStackTrace();
            String msg = "Unrecoverable exception: " + use.toString();
            context.setValue("error", msg);
        }
        catch (EboItemExistenceException iee)
        {
            iee.printStackTrace();
            String msg = "Item existence exception: " + iee.toString();
            context.setValue("error", msg);
        }
    }
}
```

5 Managing Tasks

This chapter describes how tasks work in the Content Management (CM) subsystem and explains how to reconfigure installed tasks and write and implement custom tasks. It contains the following sections:

- About tasks
- About how tasks are registered and configured
- Customizing an installed task
- Creating and implementing a new task
- Custom task sample code
- Working with task events


You also can use the CMS Administration Console to manage tasks. For more information, see Chapter 21, “Administering Automated Tasks”.

About tasks

An exteNd Director *task* is a background job or process that you can configure to run at a specified time or specified times. Typically, a task carries out a specific CM operation, such as publishing documents.

Using tasks A task must be *enabled* before it can be used in a deployed exteNd Director application. A list of enabled tasks appears in the Task section of the CMS Administration Console. You can start and stop the tasks that appear in this list while an application is running

Types of tasks There are two types of exteNd Director tasks: *periodic* and *scheduled*. Periodic tasks are configured to run at regular intervals (specified in milliseconds). Scheduled tasks are configured to run at specific dates and times. A task can be scheduled, periodic, or both.

Installed tasks

The following tasks are installed with the CM subsystem:

Task name	Description
publish	Publishes a specified set of documents.
expire	Expires a specified set of documents.
janitor	Removes a specified set of documents.
synch	Synchronizes CM data with the Search subsystem engine, which by default is based on the Autonomy Dynamic Reasoning Engine (DRE); updates to CM data are propagated to the DRE. **NOTE:** The synch task appears in the Task section only when the CM subsystem's Search synchronization mode is set to **batch**. In **immediate** synchronization mode, the CM subsystem automatically performs search synchronization operations.
default	For debugging and demonstration purposes. This task is not automatically implemented in a deployed application.

Configurability These installed tasks are highly configurable (in a set of three XML files) and can be adjusted to meet the specific needs of your application. For example, you might provide a task such as the publisher or the janitor with a query that defines the scope of its operation. Such a query would specify the set of documents on which the task was to operate.

 For information on which files you need to edit to reconfigure an installed task, see "About how tasks are registered and configured" on page 87. For an example, see "Customizing an installed task" on page 89.

Custom tasks

You may not be able to meet the needs of some applications just by reconfiguring the installed tasks. In such cases you can also create new, application-specific tasks.

When you create a task, you:

- Register its type, name, description, and configuration information
- Create Java classes to provide the task's functionality and register these classes.

📖 For information on the files you need to edit to register and configure a new task—and register the Java classes you create for it, see "About how tasks are registered and configured" next.

About how tasks are registered and configured

Tasks—and the Java classes associated with them—are registered and configured in three XML files in your project's library/ContentMgmtService/ContentMgmtService.spf/ContentMgmtService-conf directory:

XML file	What it does
tasktypes.xml	Establishes the names and descriptions of tasks and identifies them as periodic or scheduled
Default_tasklist.xml	Configures tasks
services.xml	Associates tasks (and other exteNd Director functions) with their respective Java classes

tasktypes.xml

The entries in tasktypes.xml establish the name and description of each task and identify each task as either periodic or scheduled (or both). The structure of this file must conform to **framework-task-type_3_0.dtd** in your project's library/FrameworkService/FrameworkService.spf/DTD directory.

Here is an excerpt from tasktypes.xml, showing how the file is structured and how the **default**, **synch**, and **publish** installed tasks are initially defined:

```
<framework-task-types>
<!-- PERIODIC TASK TYPES -->
   <periodic>
      <task-type>
         <type-name>default</type-name>
         <type-descr>The Default Periodic Task</type-descr>
      </task-type>
      <task-type>
         <type-name>synch</type-name>
         <type-descr>Periodic CM/Search Engine Synchronization
Task</type-descr>
      </task-type>
      <task-type>
         <type-name>publish</type-name>
         <type-descr>Periodic Document Publish Task</type-descr>
      </task-type>
```

```
      ...
   </periodic>
<!-- SCHEDULED TASK TYPES -->
   <scheduled>
   ...
   </scheduled>
</framework-task-types>
```

Default_tasklist.xml

The entries in Default_tasklist.xml configure each task in conformance with **contentmgmt-task-list_3_0.dtd** in your project's library/ContentMgmtService/ContentMgmtService.spf/DTD directory.

Here is an excerpt from Default_tasklist.xml showing how the file is structured and how the **periodic-publish** task is configured:

```
<contentmgmt-task-list>
   ...
   <periodic-publish>
      <task-name>Default Repository Document Publish</task-name>
      <description>The Default Repository Document Publish
Task</description>
      <since-last>false</since-last>
      <enabled>true</enabled>
      <interval>
         <millis>86400000</millis>
         <exact>false</exact>
      </interval>
      <do-all-not-yet-published>false</do-all-not-yet-published>
      <do-all-unpublished>false</do-all-unpublished>
      <do-all-ready>false</do-all-ready>
      <force-publish>false</force-publish>
   </periodic-publish>
   ...
</contentmgmt-task-list>
```

Naming convention Note that the *tag name* for the periodic-publish task is constructed from its type (periodic) and its name (publish) as defined in tasktypes.xml, connected by a hyphen. This is a required naming convention for the Default_tasklist.xml file.

Enabling or disabling a task Note that to enable a task, you set the content of the `<enabled>` tag to **true**. To disable a task, you set this value to **false**.

services.xml

The services.xml file includes entries that associate tasks (and other exteNd Director functions) with their respective Java classes. The structure of this file must conform to **framework-services_3_0.dtd** in your project's library/FrameworkService/FrameworkService.spf/DTD directory.

Here is an excerpt from services.xml showing how the **periodic-publish** task is handled:

```
<service>
<interface>com.sssw.cm.periodic-publish</interface>
<impl-class>com.sssw.cm.task.impl.EboDocPeriodicPublishTask</impl-
class>
<description>Periodic CM Document Publish Task</description>
<max-instances>0</max-instances>
<startup>M</startup>
<namespaced>false</namespaced>
</service>
```

Graphical view exteNd Director also provides a graphical view of this file where you can add new entries.

New tasks only You will need to add new entries to services.xml only if you create new tasks.

Customizing an installed task

You customize an installed task by editing its configuration in the Default_tasklist.xml file.

In the following example, a document query has been added to the definition of the periodic-publish task. The query is specified in the `<content-search>` element.

The added code (shown in bold) configures the periodic-publish task to publish all documents whose **STATUS** has been set to **Reviewed**:

```
<periodic-publish>
   <task-name>Default Repository Document Publish</task-name>
   <description>The Default Repository Document Publish
Task</description>
   <since-last>false</since-last>
   <enabled>true</enabled>
   <interval>
      <millis>86400000</millis>
      <exact>false</exact>
   </interval>
   <do-all-not-yet-published>false</do-all-not-yet-published>
   <do-all-unpublished>false</do-all-unpublished>
```

```
    <do-all-ready>false</do-all-ready>
    <force-publish>false</force-publish>
    <content-search>
       <where-clause>
          <eq>
             <var>STATUS</var>
             <val>Reviewed</val>
          </eq>
       </where-clause>
    </content-search>
 </periodic-publish>
```

 For a complete description of the elements and values you can use to construct a document query within a task's definition, see the definition of the `<content-search>` element in contentmgmt-task-list_4_0.dtd.

Need to redeploy You must redeploy your application EAR for any task configuration changes to take effect.

Creating and implementing a new task

The following procedure is based on the example of creating a new task named **new-doc-notifier** that checks for new documents and notifies a list of recipients about the new documents by e-mail.

➢ **To create and implement a new task:**

1. Register your task type.

   To do so, modify the tasktypes.xml file. You can register the task as scheduled, periodic, or both scheduled and periodic. In this example, the new task is periodic:

```
<periodic>
   ..............
   <task-type>
      <type-name>new-doc-notifier</type-name>
      <type-descr>Periodic CM task for notifying of any new
documents.</type-descr>
   </task-type>
```

2. Register your task in the tasklist.

   To do so, add a new element to the Default_tasklist.xml file:

```
<periodic-new-doc-notifier>
   <task-name>New Document Notifier</task-name>
   <description> Periodic CM task for notifying of any new
documents.</description>
   <since-last>false</since-last>
   <enabled>true</enabled>
```

```
   <interval>
      <millis>86400000</millis>
         <exact>false</exact>
   </interval>
   <!-- any other XML that is specific to the custom task goes
here... -->
   <!-- for instance, there may be a node here defining the list
of email recipients. -->
   <recipients>
      <recipient>user@myco.com</recipient>
      <recipient>user2@myco.com</recipient>
      <recipient>user3@myco.com</recipient>
   </recipients>
      <mail-smtp-host>smtp_host@myco.com</mail-smtp-host>
      <subject>New documents have been added</subject>
      <text>The following new documents have been added:</text>
</periodic-new-doc-notifier>
```

Naming convention Note that the name of the XML tag surrounding the task definition (`<periodic-new-doc-notifier>`) must be constructed from the task's type (periodic or scheduled) and the task's name in Default_tasklist.xml. This naming convention is required.

3 Write Java classes for the new task.

The generic exteNd Director task management API is provided in the com.sssw.fw.task.api package. This package contains very general interfaces for tasks, task types, and task management:

- EbiTask
- EbiScheduledTask
- EbiPeriodicTask
- EbiTaskType
- EbiTaskManager

The CM subsystem subclasses those interfaces in its own task management package (com.sssw.cm.task.api). It provides its own EbiTask and EbiTaskManager along with EbiTaskMgmtDelegate, all three of which should be used for managing tasks. This package also contains generic interfaces for document publishing, expiration, removal, and synchronization between the CM subsystem and the Search subsystem engine.

When writing your own custom task, you should implement one of the following interfaces:

- com.sssw.fw.task.api.EbiPeriodicTask
- com.sssw.fw.task.api.EbiScheduledTask

In the code for the new-doc-notifier example, the NewDocumentNotifier class extends com.sssw.cm.task.impl.EboTask and encapsulates the details of the task's duties and how they are carried out. The PeriodicNewDocumentNotifier class is the periodic subclass of the NewDocumentNotifier class.

📖 For a complete listing of the Java code for the new-doc-notifier example, see "Custom task sample code" on page 93.

4 Register the new task's Java class.

To do so, add an entry to the services.xml file under `<!-- Task management related objects -->`:

```
<!-- Periodic tasks -->
.....................
   <service>
      <interface>com.myco.cmtask.api.periodic-new-doc-
notifier</interface>
      <impl-
class>com.myco.cmtask.impl.PeriodicNewDocumentNotifier</impl-
class>
      <description>The periodic new document notifier
class.</description>
      <max-instances>0</max-instances>
      <startup>M</startup>
   </service>
```

Naming convention Note that in order for the object to be factoried and instantiated correctly, the interface naming should correspond to the task kind and type. For example, **periodic** and **new-doc-notifier** map to **periodic-new-doc-notifier** in the `<interface>` node value.

5 Prepare for your custom task to be loaded and instantiated correctly:

5a Place your custom task class or classes into a separate JAR.

5b Add the JAR to your exteNd Director EAR.

5c In the PMC WAR of your application, add the custom class JAR to the **Class-Path** section of the META-INF/MANIFEST.MF file.

This ensures that class loading works correctly and that users can manage the custom tasks in the Task section of the CMS Administration Console.

6 Build and deploy your application.

7 Start the task:

7a In a browser window, launch the CMS Administration Console and log in.

7b Click the **Tasks** button to enter Tasks mode.

7c In the Tasks Pane, click to select your task and then click the **Start** button.

TIP: To stop a task, click the **Stop** button.

Custom task sample code

This section provides a listing of the Java code for the NewDocumentNotifier class discussed in Step 3 above.

This section also includes the code for the PeriodicNewDocumentNotifier class, which is the periodic subclass of the NewDocumentNotifier class.

NewDocumentNotifier

```
package com.myco.cmtask.impl;

// Java imports
import java.io.*;
import java.sql.Timestamp;
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

// FW imports
import com.sssw.fw.api.*;
import com.sssw.fw.exception.*;
import com.sssw.fw.log.*;
import com.sssw.fw.task.exception.*;
import com.sssw.fw.util.*;

// CM imports
import com.sssw.cm.api.*;
import com.sssw.cm.factory.*;
import com.sssw.cm.task.api.*;
import com.sssw.cm.task.impl.EboTask;

// Other imports
import org.w3c.dom.*;

abstract public class NewDocumentNotifier extends EboTask
{
    //
    // Constants
    //

    protected static final String RECIPIENTS = "recipients";
    protected static final String RECIPIENT = "recipient";
    protected static final String SMTP_HOST = "mail-smtp-host";
    protected static final String SUBJECT = "subject";
    protected static final String TEXT = "text";
    protected static final String SENDER = "sender";
    protected static final String NEWLINE = "\n";
    protected static final String MAIL_SMTP_HOST =
"mail.smtp.host";
    protected static final String LINE_SEPARATOR =
"line.separator";

    // These actually belong in a resource bundle...
    protected static final String ERROR = "An error occurred while
executing the New Document Notifier task.";
    protected static final String DEFAULT_SUBJECT = "New documents
have been added";
    protected static final String DEFAULT_TEXT = "The following
documents have been added:";
```

```
    protected static final String DEFAULT_SENDER =
"notifier@myco.com";
    protected static final String LOCATION = "Location: ";
    protected static final String TITLE = "Title: ";
    protected static final String AUTHOR = "Author: ";

    //
    // Member variables
    //

    protected EbiLog m_log;                  // Our log
    protected ArrayList m_recipients;        // Notification
recipients
    protected String m_smtpHost;             // SMTP host
    protected String m_subject;              // Message subject
    protected String m_text;                 // Message text
    protected String m_sender;               // Sender
    protected String m_lineSep;              // Line separator

    // Constructor
    public NewDocumentNotifier()
    {
        // Use the CM log
        m_log = EboLogFactory.getLog(EboLogFactory.CM);

        m_recipients = new ArrayList();

        m_subject = DEFAULT_SUBJECT;

        m_text = DEFAULT_TEXT;

        m_sender = DEFAULT_SENDER;
    }

    // Initialization from XML
    public void fromXML(Node node)
    {
        // Rely on the superclass to get all the general task
        // settings
        super.fromXML(node);

        try
        {
            NodeList nodes = node.getChildNodes();
            if (nodes != null)
            {
                // Process the nodes
                for (int i = 0; i < nodes.getLength(); i++)
                {
                    Node child = nodes.item(i);
                    String nodeName = child.getNodeName();

                    if (child.getNodeType() == Node.ELEMENT_NODE)
                    {
```

```
                        // Recipient list
                        if (RECIPIENTS.equals(nodeName))
                            processRecipientList(child);

                        // SMTP host
                        else if (SMTP_HOST.equals(nodeName))
                            m_smtpHost = getElementValue(child);

                        // Message subject
                        else if (SUBJECT.equals(nodeName))
                            m_subject = getElementValue(child);

                        // Base message text
                        else if (TEXT.equals(nodeName))
                            m_text = getElementValue(child);

                        // Sender
                        else if (SENDER.equals(nodeName))
                            m_sender = getElementValue(child);
                    }

                } // End for each node
            }
        }
        catch (Exception ex)
        {
            EboExceptionHelper.handleException(
                ex,         // The exception
                m_log,      // Our log to write exception into
                false,      // Don't print stack trace to console
                false);     // Don't rethrow as a runtime exception
        }
    }

    // Process the list of recipients provided in the XML task
definition
    protected void processRecipientList(Node node)
    {
        NodeList nodes = node.getChildNodes();
        if (nodes != null)
        {
            // Process the nodes
            for (int i = 0; i < nodes.getLength(); i++)
            {
                Node child = nodes.item(i);

                if (child.getNodeType() == Node.ELEMENT_NODE)
                {
                    String nodeName = child.getNodeName();
                    if (RECIPIENT.equals(nodeName))
                    {
                        String recipient = getElementValue(child);
                        if (!EboStringMisc.isEmpty(recipient))
                            m_recipients.add(recipient);
```

```
                    }
                }
            }
        }
    }

    // Extract a node value from a Node
    public static String getElementValue(Node node)
    {
        // Entities are often considered separate text nodes;
        // for example, Jim&apos;s wagon is represented by three
        // text nodes "Jim", "&apos;",and "s wagon". Thus all
        // children need to be concatenated in order to retrieve
        // the proper text node value.

        String nodeValue;
        if (node.hasChildNodes())
        {
            Node curNode = node.getFirstChild();
            nodeValue = EboStringMisc.m_emptyStr;
            while (curNode != null)
            {
                nodeValue = nodeValue + curNode.getNodeValue();
                curNode = curNode.getNextSibling();
            }
        }
        else
            nodeValue = EboStringMisc.m_emptyStr;
        return nodeValue;
    }

    // Carry out the task
    public void doTask() throws EboTaskException
    {
        try
        {
            super.doTask();

            EbiContentManager cmgr =
EboFactory.getDefaultContentManager();
            EbiDocQuery query =
(EbiDocQuery)cmgr.createQuery(EbiDocQuery.DOC_QUERY);

            // If we're to only get the data that's changed since
            // The time that the task was last run
            if (getSinceLast())
            {
                // Figure out the start of the interval
                Timestamp fromTime = getFromTime();

                // Figure out the end of the interval
                Timestamp toTime = new Timestamp((new
Date()).getTime());
```

```
                EbiQueryExpression expr = null;
                EbiQueryExpression expr2 = null;

                // Augment the where clause with the time interval
                if (fromTime != null)
                    expr = query.whereCreateDate(fromTime,
EbiDocQuery.ROP_GREATER, false);
                if (toTime != null)
                    expr2 = query.whereCreateDate(toTime,
EbiDocQuery.ROP_LEQ, false);

                // Set the augmented where clause into the query
                if (expr != null && expr2 != null)
                {
                    expr.andExpression(expr2);
                    query.setWhere(expr);
                }
            }
            // Otherwise, we'll process all the documents

            // Get the list of documents
            Collection documents =
cmgr.findElementsFiltered(m_context, query);

            // Send the e-mail notifications
            sendNotifications(documents);
        }
        catch (Exception ex)
        {
            throw new
com.sssw.fw.task.exception.EboTaskException(ex, ERROR);
        }
    }

    // Send the e-mail notifications to our recipients
    protected void sendNotifications(Collection documents)
        throws EboUnrecoverableSystemException,
EboSecurityException,
            MessagingException
    {
        if (!documents.isEmpty())
        {
            String msgText = getEmailMessageBody(documents);

            // For each recipient
            for (int i = 0; i < m_recipients.size(); i++)
            {
                String recipient = (String)m_recipients.get(i);
                send(
                    m_sender,    // From
                    recipient,   // To
                    m_smtpHost,  // Host
                    m_subject,   // Subject
                    msgText);    // Yext
```

```
            }
        }
    }

    // Generate an e-mail
    // "The following documents have been added:
    //
    // <doc 1>
    // <doc 2>
    // .......
    // <doc N>"
    protected String getEmailMessageBody(Collection documents)
        throws EboUnrecoverableSystemException,
EboSecurityException
    {
        String lineSeparator = getLineSeparator();
        StringBuffer buf = new StringBuffer(m_text);
        buf.append(lineSeparator);
        buf.append(lineSeparator);

        Iterator iter = documents.iterator();
        while (iter.hasNext())
        {
            EbiDocument doc = (EbiDocument)iter.next();
            buf.append(getDocumentDescriptor(doc));
            buf.append(lineSeparator);
            buf.append(lineSeparator);
        }

        return buf.toString();
    }

    // Send an e-mail
    protected static void send(
        String from,
        String to,
        String host,
        String subject,
        String msgText)
        throws MessagingException
    {
        Properties props = System.getProperties();
        props.put(MAIL_SMTP_HOST, host);
        Session session = Session.getDefaultInstance(props, null);

        // Create a message
        Message msg = new MimeMessage(session);
        msg.setFrom(new InternetAddress(from));
        InternetAddress[] address = { new InternetAddress(to) };
        msg.setRecipients(Message.RecipientType.TO, address);
        msg.setSubject(subject);
        msg.setSentDate(new Date());
        msg.setText(msgText);
        Transport.send(msg);
```

```
    }

    // Generate a document descriptor
    // Location: <...>
    // Title: <...>
    // Author: <...>
    protected String getDocumentDescriptor(EbiDocument doc)
        throws EboUnrecoverableSystemException,
EboSecurityException
    {
        String lineSeparator = getLineSeparator();
        StringBuffer buf = new StringBuffer(LOCATION);
        buf.append(doc.getURL(false));
        buf.append(lineSeparator);
        buf.append(TITLE);
        buf.append(doc.getTitle());
        buf.append(lineSeparator);
        buf.append(AUTHOR);
        buf.append(doc.getAuthor());
        return buf.toString();
    }

    // Figure out the line separator to use
    protected String getLineSeparator()
    {
        if (m_lineSep == null)
            m_lineSep = System.getProperty(LINE_SEPARATOR,
NEWLINE);
        return m_lineSep;
    }

    abstract protected Timestamp getFromTime();
}
```

PeriodicNewDocumentNotifier

```
package com.myco.cmtask.impl;

// Java imports
import java.sql.Timestamp;

// Framework imports
import com.sssw.fw.task.api.*;
import com.sssw.fw.task.impl.*;

// CM imports
import com.sssw.cm.api.*;
import com.sssw.cm.task.api.*;

// Other imports
import org.w3c.dom.*;

public class PeriodicNewDocumentNotifier
```

```
    extends NewDocumentNotifier
    implements EbiPeriodicTask
{
    //
    // Protected data
    //

    protected long      m_interval; // Interval, if any
    protected boolean   m_exact;    // Run asap or x millis after
                                      // current time

    //
    // Constructor
    //

    public PeriodicNewDocumentNotifier()
    {
    }

    public boolean isExact()
    {
        return m_exact;
    }

    public long getInterval()
    {
        return m_interval;
    }

    public void setExact(boolean exact)
    {
        m_exact = exact;
    }

    public void setInterval(long millis)
    {
        m_interval = millis;
    }

    public void fromXML(Node node)
    {
        super.fromXML(node);
        EboTaskHelper.getPeriodicDataFromXML(this, node);
    }

    public String toString()
    {
        return super.toString() +
            ", Interval (millis)=" + m_interval +
            ", Exact=" + m_exact;
    }

    protected Timestamp getFromTime()
    {
```

```
            // For an interval-based task, the 'from' time is 'none' if
            // the task has not run once yet; otherwise it's
            // task_first_scheduled_time + interval*times_task_ran
            return (m_timesRan < 1) ? null :
                    new Timestamp(
                        m_launchTime.getTime() + m_interval *
(m_timesRan - 1));
        }
}
```

Working with task events

Task events are an extension of the exteNd Director event model framework, consisting of *state change events*, *event producers*, and *event listeners* (including vetoable listeners). This section includes these topics:

- Task event types
- Registering for a task event
- Enabling or disabling a task event

This section assumes familiarity with exteNd Director event model and event handling. For more information, see the section on working with events in *Developing exteNd Director Applications*.

Task event types

The API defines a set of state change events related to task management operations. Event IDs are exposed on the individual event classes as well as on the com.sssw.fw.task.event.api.EbiConstants interface:

Task operation	Event ID constant
Task added	EVENT_ID_TASK_ADDED
Task completed	EVENT_ID_TASK_COMPLETED
Task disabled	EVENT_ID_TASK_DISABLED
Task enabled	EVENT_ID_TASK_ENABLED
Task failed	EVENT_ID_TASK_FAILED
Task started	EVENT_ID_TASK_STARTED
Task stopped	EVENT_ID_TASK_STOPPED
Tasks listed	EVENT_ID_TASKS_LISTED

Generic state change events In addition, there are generic state change constants representing types of changes defined in com.sssw.fw.event.api.EboStateChangeEvent.

Registering for a task event

➢ **To register a task event listener:**

- Use either the addStateChangeListener() or the addVetoableStateChangeListener method on the task manager object (com.sssw.cm.task.api.EbiTaskMgmtDelegate).

You can register for a specified type or types of events using this version of addStateChangeListener():

```
public boolean addStateChangeListener(
   BitSet events, EbiStateChangeListener listener)
```

where *events* is a bit set of event IDs.

Use the event IDs specified in com.sssw.fw.event.api.EbiConstants. For example, this code registers for the task started, stopped, and completed operations:

```
EbiTaskMgmtDelegate tmgr = new EbiTaskMgmtDelegate();
EbiStateChangeListener listener = new EbiStateChangeListener();
// Instantiate a Java BitSet and populate it
BitSet events = new BitSet();
events.set(EbiConstants.EVENT_ID_TASK_STARTED);
events.set(EbiConstants.EVENT_ID_TASK_STOPPED);
events.set(EbiConstants.EVENT_ID_TASK_COMPLETED);
// add listener
tmgr.addStateChangeListener(events, listener);
```

Enabling or disabling a task event

➢ **To enable or disable task events:**

1. Open the config.xml for the Framework subsystem in your exteNd Director project.
2. Find this property:

   ```
   com.sssw.fw.task.events.enable
   ```

3. Set the value to **true** for enable or **false** for disable.
4. Redeploy your project.

6 Managing Content Caching

This chapter describes caching in the Content Management (CM) subsystem and includes these topics:

- About caching in CM
- Summary of CM caching information
- Controlling caching in the DAC

About caching in CM

Several CM elements are cached by default while an exteNd Director application is running. Caching can increase the efficiency of an application (because the application makes fewer SQL queries of the database).

For most of these elements, you can configure your exteNd Director EAR to override the default settings for caching.

NOTE: If you make any changes to the caching settings, you must redeploy your exteNd Director EAR for the changes to take effect.

Summary of CM caching information

Caching behavior

The table below provides the following information on caching behavior for several elements of documents in the CM subsystem:

- **Element**—The name of the CM element
- **CM API object name**—The com.sssw.cm.api interface name that corresponds to the CM element
- **CM cache holder used**—The name of the cache holder used for the element
- **Default behavior**—The default caching behavior for the element
- **Setting in the CM config.xml file**—The setting to change to alter the default behavior

📖 For information, see the section on reconfiguring your EAR project in *Developing exteNd Director Applications*.

Element	CM API object name	CM cache holder used	Default behavior	Setting in CM config.xml
Extension metadata fields	EbiDocField	ContentMgmtService/-CacheHolder/Fields	Cached	com.sssw.cm.cacheFields
Document types	EbiDocType	ContentMgmtService/-CacheHolder/DocTypes	Cached	com.sssw.cm.cacheDocTypes
Folders	EbiDocFolder	ContentMgmtService/-CacheHolder/Folders	Cached	com.sssw.cm.cacheFolders
Categories	EbiDocCategory	ContentMgmtService/-CacheHolder/Categories	Cached	com.sssw.cm.cacheCategories
Document metadata	EbiDocument	ContentMgmtService/-CacheHolder/Documents	Always cached	None
Document contents	EbiDocContent	None	Never cached	None
Document content versions	EbiDocVersion	None	Never cached	None

Caching of folders, categories, and document metadata

Folders, categories, and document metadata, when cached, are cached by both UUID and URL.

Document metadata is always cached.

About document content and versions

Document contents and versions of document content are not cached, because some cached content might require excessively large amounts of memory.

Controlling caching in the DAC

Another place you can control the caching process is on the Cache tab of the Configuration section of the DAC. For example, you can flush a single cache or all caches at once. This can be helpful when you are doing diagnostic work on a running exteNd Director application.

 For more information on controlling caching in the DAC, see the discussion of the Cache tab in *Developing exteNd Director Applications*.

7 Importing and Exporting Content

This chapter describes the import and export facilities provided with the exteNd Director Content Management (CM) subsystem. It has these sections:

- About importing and exporting
- About the export facility
- About the import facility
- Customizing imports and exports

About importing and exporting

The CM subsystem includes facilities for importing and exporting data between databases or within a single database.

Uses for the import and export facilities include:

- Moving or copying folders, categories, and documents within a repository
- Moving CM data between different stages of development
- Integrating with third-party vendors
- Backing up and restoring CM data
- Debugging and data analysis

This chapter describes how the import and export functions work and how you can customize them.

Using the import/export facilities

📖 You can use the import/export facilities in the CMS Administration Console to import and export content.

📖 For more information, see Chapter 20, "Importing and Exporting Content".

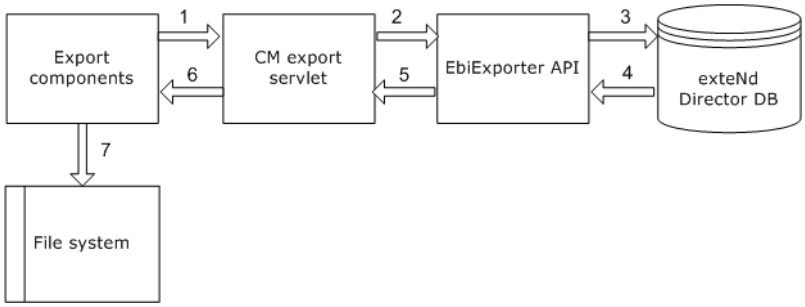
About the export facility

The export facility comprises these elements:

Export facility element	Description
Export component(s)	A portal portlet (or other UI element) that gets the data export descriptor (DED) selected by the user and lets the user save the CM archive generated by the EbiExporter object. **NOTE:** This function is provided in the CMS Administration Console.
Data export descriptor (DED)	An XML descriptor that defines the export data selected by the user. 📖 For more information, see "Customizing the data export descriptor (DED)" on page 113.
EbiExporter	Contains the API for exporting CM data. This object queries the CM repository for data based on the scope specified in the DED and then packages the result into a CM archive.
CM export servlet	Provides a connection between the EbiExporter object and the exporter portlet. This object passes the DED from the portlet to the exporter API and then gets the CM archive from the exporter and returns it to the portlet.
CM archive	A ZIP file that contains the CM export data and a default data import descriptor (DID) for subsequent use with the import facility. 📖 For a description of the archive contents, see "Structure of the data import or export archive" on page 297.

Export process

Here is how the export process works:



1. The export process begins when a portlet gets the selected DED and passes it to the CM export servlet.
2. The export servlet forwards the DED to an object that implements the EbiExporter API.

   NOTE: The servlet provides remote access to the EbiExporter API; however, the portlet could call the API directly.
3. The CM exporter API uses the DED to create a query representing all data and infrastructure the user has identified and executes the query against the CM database.
4. The CM database responds to the query by returning a raw result set.
5. The CM exporter API formats the raw query result into a structured ZIP archive file containing the content and data descriptors and returns the archive to the export servlet.
6. The export servlet returns the ZIP file to the portlet as bytes of content with a MIME type of application/zip.
7. The export portlet gets the ZIP file and saves it in a disk location specified by the user.

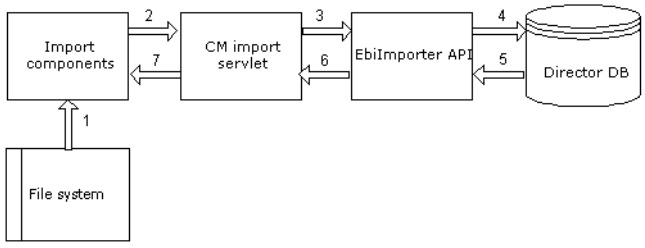
About the import facility

The import facility is made up of these elements:

Import facility element	Description
Import component(s)	A portal portlet (or other UI element) that provides the UI for the user to select import data. This import function is provided in the CMS Administration Console.
Data import descriptor (DID)	An XML descriptor based on the contents of the CM archive generated by the exporter. 📖 For more information, see "Customizing the data import descriptor (DID)" on page 114.
EbiImporter	Contains the API for importing CM data. It extracts the DID from the CM archive and uses it to insert data into the target database.
CM import servlet	Provides a connection between the EbiImporter and the CMS Administration Console. This object passes the CM archive to the importer API and creates an XML document that enumerates any warnings and failures encountered in the import process.

Import process

Here is how the import process works:



1. The import process begins when a portlet allows the user to select the CM archive containing the DID.

   The CM archive is generated by the export facility.

2. The import portlet posts the ZIP file to the CM import servlet via HTTP.

   NOTE: The servlet provides remote access to the EbiImporter API; however, the portlet could call the API directly.

3. The import servlet passes the ZIP file to the CM importer as an input stream.

4 The CM importer extracts the DID from the ZIP and uses it to transfer the data from the ZIP file to the underlying content repository.

5 The CM importer returns a summary of its execution to the CM import servlet.

6 The import servlet creates an XML document that represents successes, warnings, and failures that were encountered during the import process.

7 The import servlet returns the XML document to the portlet, which generates a report for the user.

Customizing imports and exports

You can customize data imports and exports several ways:

Operation	Customizing option
For an export	Providing a custom DED file before executing an export
For an import	Editing the generated DID before executing an import
	Adding or deleting items from the CM archive before executing an import
For an export or an import	Providing your own logic by implementing the import and export APIs

Customizing the data export descriptor (DED)

The data export descriptor (DED) is an XML file you can use to:

- Set configuration options for data export
- Specify what CM data will be placed in your export archive

Format for entries The entries in your DED must conform to the DTDs in your project's library/ContentMgmtService/ContentMgmtService.spf/DTD directory.

Sample DED files There are several sample DED files in your project's library/ContentMgmtService/ContentMgmtService.spf/DTD directory. Each of these samples represents a typical export scenario:

Sample file	Demonstrates how to
contentmgmt-export-descr_5_0_sample.xml	◆ Export all data out of the CM system ◆ Specify options for the operation

Sample file	Demonstrates how to
contentmgmt-export-descr_5_0_sample2.xml	◆ Export specific infrastructural data out of the CM system
contentmgmt-export-descr_5_0_sample3.xml	◆ Export specific infrastructural data out of the CM system ◆ Export dependent items in the element hierarchy
contentmgmt-export-descr_5_0_sample4.xml	◆ Export all documents and supporting infrastructure using default configuration options
contentmgmt-export-descr_5_0_sample5.xml	◆ Export certain elements of documents that satisfy a query conforming to **contentmgmt-docmeta-search_5_0.dtd**

Customizing the data import descriptor (DID)

The data import descriptor (DID) is an XML file you can use to:

- Set configuration options for data import
- Set overwrite options for each type of CM data
- Specify the target folder for the import

Format for entries The entries in your DID must conform to **contentmgmt-import-descr_5_0.dtd** in your project's library/ContentMgmtService/ContentMgmtService.spf/DTD directory.

Sample DID file There is a sample DID file in your project's library/ContentMgmtService/ContentMgmtService.spf/DTD directory that shows how to set all available import options:

- **contentmgmt-import-descr_5_0_sample.xml**

Accessing the import and export API

In most cases, editing the descriptors or the CM archive should provide most of the flexibility you need. However you can access the import and export API directly for tasks ranging from ad hoc imports and exports to writing your own facility.

Potential uses of the functionality provided by EbiImporter and EbiExporter include:

- Importing preexisting documents into a new exteNd Director CM system
- Backing up and restoring CM data
- Replicating data between two CM systems
- Moving and copying data within a single CM system

 For more information, see the online API documentation for EbiExporter and EbiImporter.

8 Working with Content Management Events

This chapter describes how to handle events related to Content Management (CM) subsystem operations and activities. It has these sections:

- About CM events
- Registering for CM events
- Enabling CM events

 This chapter assumes familiarity with exteNd Director event model and event handling. For more information, see the chapter on working with events in *Developing exteNd Director Applications*.

About CM events

CM events are an extension of the exteNd Director event model framework, consisting of *state change events*, *event producers*, and *event listeners* (including vetoable listeners). The API for CM events is defined in these packages:

- com.sssw.cm.event.api
- com.sssw.cm.event.util

CM event types

The API defines a set of state change events related to CM operations on documents, folders, and other elements—as well as general activities like data import/export. Event IDs are exposed on the individual event classes as well as on the com.sssw.cm.event.api.EbiConstants interface. In addition, there are state change constants defined in com.sssw.fw.event.api.EboStateChangeEvent.

Here is a list of events defined for the CM subsystem:

Event type	Operation	Event constant
Category	added	EVENT_ID_CATEGORY_ADDED
	contents listed	EVENT_ID_CATEGORY_CONTENTS_LISTED
	copied	EVENT_ID_CATEGORY_COPIED
	moved	EVENT_ID_CATEGORY_MOVED
	removed	EVENT_ID_CATEGORY_REMOVED
	document removed	EVENT_ID_DOC_REMOVED_FROM_CATEGORY
	metadata retrieved	EVENT_ID_CATEGORY_RETRIEVED
	metadata updated	EVENT_ID_CATEGORY_UPDATED
Data export/import	data exported	EVENT_ID_DATA_EXPORTED
	data imported	EVENT_ID_DATA_IMPORTED

Event type	Operation	Event constant
Document	added	EVENT_ID_DOC_ADDED
	added to category	EVENT_ID_DOC_ADDED_TO_CATEGORY
	checked in	EVENT_ID_DOC_CHECKED_IN
	checked out	EVENT_ID_DOC_CHECKED_OUT
	copied	EVENT_ID_DOC_COPIED
	moved	EVENT_ID_DOC_MOVED
	published	EVENT_ID_DOC_PUBLISHED
	removed	EVENT_ID_DOC_REMOVED
	retrieved	EVENT_ID_DOC_RETRIEVED
	rolled back	EVENT_ID_DOC_ROLLED_BACK
	unchecked out	EVENT_ID_DOC_UNCHECKED_OUT
	unlocked	EVENT_ID_DOC_UNLOCKED
	unpublished	EVENT_ID_DOC_UNPUBLISHED
	updated	EVENT_ID_DOC_UPDATED
	link added	EVENT_ID_DOC_LINK_ADDED
	link removed	EVENT_ID_DOC_LINK_REMOVED
	link retrieved	EVENT_ID_DOC_LINK_RETRIEVED
	link updated	EVENT_ID_DOC_LINK_UPDATED
	links listed	EVENT_ID_DOC_LINKS_LISTED
Document type	added	EVENT_ID_DOC_TYPE_ADDED
	removed	EVENT_ID_DOC_TYPE_REMOVED
	retrieved	EVENT_ID_DOC_TYPE_RETRIEVED
	updated	EVENT_ID_DOC_TYPE_UPDATED
	listed	EVENT_ID_DOC_TYPES_LISTED
	fields listed	EVENT_ID_DOC_TYPE_FIELDS_LISTED

Event type	Operation	Event constant
Field	added	EVENT_ID_DOC_FIELD_ADDED
	added to document type	EVENT_ID_DOC_FIELD_ADDED_TO_TYPE
	listed	EVENT_ID_DOC_FIELDS_LISTED
	removed	EVENT_ID_DOC_FIELD_REMOVED
	removed from document type	EVENT_ID_DOC_FIELD_REMOVED_FROM_TYPE
	retrieved	EVENT_ID_DOC_FIELD_RETRIEVED
	updated	EVENT_ID_DOC_FIELD_UPDATED
Folder	added	EVENT_ID_FOLDER_ADDED
	contents listed	EVENT_ID_FOLDER_CONTENTS_LISTED
	copied	EVENT_ID_FOLDER_COPIED
	moved	EVENT_ID_FOLDER_MOVED
	removed	EVENT_ID_FOLDER_REMOVED
	retrieved	EVENT_ID_FOLDER_RETRIEVED
	updated	EVENT_ID_FOLDER_UPDATED
Layout document descriptor	added	EVENT_ID_LLD_ADDED
	listed for a style	EVENT_ID_LLDS_LISTED
	removed	EVENT_ID_LLD_REMOVED
	retrieved	EVENT_ID_LLD_RETRIEVED
	updated	EVENT_ID_LLD_UPDATED
Layout style	added	EVENT_ID_DOC_LAYOUT_STYLE_ADDED
	removed	EVENT_ID_DOC_LAYOUT_STYLE_REMOVED
	retrieved	EVENT_ID_DOC_LAYOUT_STYLE_RETRIEVED
	updated	EVENT_ID_DOC_LAYOUT_STYLE_UPDATED
	styles listed	EVENT_ID_DOC_LAYOUT_STYLES_LISTED
Directory entry lookup	by absolute path (URL)	EVENT_ID_LOOKUP_BY_ABSOLUTE
	by ancestor and relative path	EVENT_ID_LOOKUP_BY_RELATIVE

Event type	Operation	Event constant
CM repository	added	EVENT_ID_REPOSITORY_ADDED
	listed	EVENT_ID_REPOSITORIES_LISTED
	removed	EVENT_ID_REPOSITORY_REMOVED
	retrieved	EVENT_ID_REPOSITORY_RETRIEVED
	updated	EVENT_ID_REPOSITORY_UPDATED
Query/search	document query executed	EVENT_ID_DOC_QUERY_EXECUTED
	document search query executed	EVENT_ID_DOC_SEARCH_QUERY_EXECUTED
Security	access checked	EVENT_ID_ACCESS_CHECKED
	admin access checked	EVENT_ID_ADMIN_ACCESS_CHECKED
	access removed	EVENT_ID_SECURITY_REMOVED
	access retrieved	EVENT_ID_SECURITY_RETRIEVED
	access set	EVENT_ID_SECURITY_SET

Registering for CM events

This section includes these sections:

- Registering for events on directory elements
- Specifying event types
- Using the event helper class
- Event registration examples

Registering for events on directory elements

Event support in the CM subsystem provides convenience methods for registration of listeners based on CM categories, folders, and documents. The methods are available on an EbiContentMgmtDelegate object:

Listener convenience method	What it subscribes to
addCategoryStateChangeListener()	All category events
addDocumentStateChangeListener()	All document events

Listener convenience method	What it subscribes to
addFolderStateChangeListener()	All folder events
addVetoableCategoryStateChangeListener()	All category events, with ability to veto operation
addVetoableDocumentStateChangeListener()	Subscribes to all document events, with ability to veto operation
addVetoableFolderStateChangeListener()	Subscribes to all folder events, with ability to veto operation

For example, here is how to subscribe to all events that relate to folder operations:

```
EbiContentMgmtDelegate cmgr =
  com.sssw.cm.client.EboFactory.getDefaultContentMgmtDelegate();
cmgr. addFolderStateChangeListener (myStateChangeListener);
```

Specifying event types

You can register for specified type(s) of events using the framework version of addStateChangeListener(), available on EbiContentMgmtDelegate:

```
public boolean addStateChangeListener(
   BitSet events, EbiStateChangeListener listener)
```

where *events* is a bit set of event IDs. The CM API provides some helper methods for specifying a bit set, as described in "Using the event helper class" next.

You can also filter events that occur on either a specific directory entry or a directory and entries underneath it (recursively). In order to register for events that occur within a certain directory entry scope, add the listener using this method:

```
public boolean addStateChangeListener(
   BitSet events, EbiDirectoryEntry entry, int depth,
       EbiStateChangeListener listener)
```

Method parameter	What it means
events	Bit set of event IDs
entry	Directory entry (a folder, a category, or a document)

Method parameter	What it means
depth	How deep event tracking should go: ◆ **0** means that state changes that occur only on the entry itself ◆ **1** means that state changes that occur to the entry and its children ◆ **-1** means that state changes that occur to the entry and any of its descendant Any other depth specifies that state changes that occur on the entry and its descendants to that depth in the entry hierarchy are to be tracked
listener	A new listener object

Using the event helper class

The com.sssw.cm.event.util.EboEventHelper class provides utilities for managing event sets. It includes these methods:

Event helper method	What it does
getFullEventIDSet()	Returns a bit set containing the full set of CM events exposed on all CM element types
getEventIDSet(String elType)	Returns a bit set containing the full set of CM events exposed on a specified CM element type
getEventIDSet(int stateChangeID)	Returns a bit set for all events that map to a given state change type
adjustEventIDSet()	Given a bit set for event IDs, turns on or off the bits for CM events of the specified state change type

Event registration examples

Listen on one event for all elements This example adds a listener for the "create" state change event on all elements:

```
EbiContentMgmtDelegate cmgr =
   com.sssw.cm.client.EboFactory.getDefaultContentMgmtDelegate();

BitSet events = EboEventHelper.getEventIDSet(
  com.sssw.fw.event.api.EboStateChangeEvent.SC_CREATE);
cmgr.addStateChangeListener(events, MyListener);
```

Listen on all events for two element types This example adds a listener for all changes on document types and fields only; note the use of adjustEventIDSet():

```
EbiContentMgmtDelegate cmgr =
   com.sssw.cm.client.EboFactory.getDefaultContentMgmtDelegate();

BitSet events =
   EboEventHelper.getEventIDSet(EbiDocType.EL_DOC_TYPE);
EboEventHelper.adjustEventIDSet(events,
   EbiDocField.EL_DOC_FIELD, true);
cmgr.addStateChangeListener(events, Mylistener);
```

Listen on multiple events for all elements This example adds event types by instantiating a new bit set; this is the technique to use for specifying multiple sets of events:

```
EbiContentMgmtDelegate cmgr =
   com.sssw.cm.client.EboFactory.getDefaultContentMgmtDelegate();

BitSet events = new BitSet();
events.set(com.sssw.cm.event.api.EbiConstants.
  EVENT_ID_ACCESS_CHECKED);
events.set(com.sssw.cm.event.api.EbiConstants.
  EVENT_ID_ADMIN_ACCESS_CHECKED);
events.set(com.sssw.cm.event.api.EbiConstants.
  EVENT_ID_SECUIRTY_RETRIEVED);

cmgr.addStateChangeListener(events, Mylistener);
```

Listen on all events except for a specified element type This example shows how to use the boolean argument on adjustEventIDSet() to turn off an event set:

```
EbiContentMgmtDelegate cmgr =
   com.sssw.cm.client.EboFactory.getDefaultContentMgmtDelegate();

BitSet events = EboEventHelper.getFullEventIDSet();
EboEventHelper.adjustEventIDSet(events, EbiDocType.EL_DOC_TYPE,
false);
cmgr.addStateChangeListener(events, Mylistener);
```

Enabling CM events

➢ **To enable or disable CM events:**

1. Open config.xml for CM your exteNd Director project.
2. Find this property:

   ```
   com.sssw.cm.events.enable.Default
   ```

3. Set the value to **true** for enable or **false** for disable.
4. Redeploy your project.

II WebDav

Describes how to set up and use a WebDav client with the Content Management (CM) subsystem

- Chapter 9, “Using WebDAV Clients with exteNd Director for Collaborative Authoring”
- Chapter 10, “Building Your Own WebDAV Client”
- Chapter 11, “Working with WebDAV Events”

9 Using WebDAV Clients with exteNd Director for Collaborative Authoring

This chapter describes the exteNd Director WebDAV subsystem, which provides support for the Web Distributed Authoring and Versioning (WebDAV) communications protocol. Using this protocol, the WebDAV subsystem allows you to access server-side content in the exteNd Director Content Management (CM) subsystem from third-party or custom WebDAV client applications.

This chapter includes the following topics:

- What is WebDAV?
- About exteNd Director's WebDAV support
- Installing the exteNd Director WebDAV subsystem
- Deploying the exteNd Director WebDAV subsystem
- Setting up the client
- Supported WebDAV methods
- Public WebDAV server

What is WebDAV?

The *WebDAV protocol* extends the Hypertext Transfer Protocol (HTTP) to support asynchronous collaborative authoring on the Web.

As the standard protocol that allows Web browsers to communicate with Web servers, HTTP has transformed the Web into a readable medium by allowing users to view and download individual static documents as read-only information. However, HTTP falls short of supporting write operations such as simultaneous editing of multiple resources on the Web.

WebDAV goes the next step by providing extensions to HTTP that create a distributed writable Web environment. Using WebDAV, multiple users can create content locally or remotely using WebDAV-enabled authoring tools, then save content directly to an URL on an HTTP server.

This section provides a brief overview of WebDAV.

📖 For more detailed information on WebDAV, search on the Web for **rfc2518**—the WebDAV specification. The following URL provided helpful information at the time this chapter was published:

- http://asg.web.cmu.edu/rfc/rfc2518.html

Information elements for distributed Web authoring

The WebDAV protocol provides methods that act on Web *resources*, *collections*, and *properties*—key information elements used in distributed Web authoring:

Element	Definition	Examples
Resource	Any piece of information that is stored on a Web server and whose location is described by an URL	Web pages, documents, and bitmap images
Collection	A resource that serves as a container for other resources, including other collections. Collections provide a paradigm for grouping and searching resources	Folders and directories
Property (metadata)	Descriptive information that is associated with Web resources but not stored as part of their content	Author, title, publication date, and expiration date

WebDAV extensions to HTTP

The WebDAV protocol provides extensions to HTTP through a set of open standards that can be used by any distributed authoring tool. These extensions support the following key requirements for collaborative authoring on the Web:

Authoring requirement	How WebDAV meets the requirement
Overwrite protection	Mediates concurrent access to content by multiple authors by providing resource locking for write operations
Properties	Provides methods for creating, modifying, reading, and deleting properties

Authoring requirement	How WebDAV meets the requirement
Namespace manipulation	Supports copying and moving multiple Web resources by manipulating names and directories within the namespaces of URLs
Collections	Provides methods for creating and deleting collections, adding members to a collection, removing members from a collection, and listing members of a collection
Version management	Supports the storage of resource revisions for later retrieval; automatic versioning records successive modifications to a resource
Access control	Limits the access rights of a particular authenticated principal to a given resource

About exteNd Director's WebDAV support

The exteNd Director WebDAV subsystem is designed to work with any WebDAV-compliant client application.

Works with WebDAV-compliant authoring tools When you install the exteNd Director WebDAV subsystem, you can create content in your preferred WebDAV-compliant authoring tool and still take advantage of the standard document management capabilities of the exteNd Director CM subsystem on your server—functions such as checkin, checkout, and versioning.

Includes WebDAV client API While most third-party WebDAV clients support these standard document management functions, they do not support the more sophisticated features of the CM subsystem, such as categorization and document creation using custom templates. To bridge this gap, the WebDAV subsystem also includes a WebDAV client API that provides classes and methods for accessing these custom features from your own client applications.

📖 For more information about the WebDAV client API, see Chapter 10, "Building Your Own WebDAV Client".

What you can do with the exteNd Director WebDAV subsystem

When you install the WebDAV subsystem, you will be able to perform the following functions remotely from your WebDAV client application:

- **Save** your content in the content repository
- **Get** the latest version of your content from the content repository for editing
- **Lock** content for editing in the content repository and know that your changes will not be overwritten by another author
- **Unlock** content so that it is available to other authors for editing
- **Copy and move** content across collections within the hierarchical physical infrastructure of the content repository
- **Delete** content from the content repository
- **Make** new collections in the content repository
- **Retrieve** resources and collections from the server
- **Upload** resources and collections from the client to the server

WebDAV-enabled clients implement these functions in different ways. Consult your client documentation to learn how to use specific third-party tools with the WebDAV protocol.

📖 For more information about the WebDAV methods exteNd Director supports, see "Supported WebDAV methods" on page 134.

How exteNd Director stores content from WebDAV clients

When you save content created using a third-party WebDAV client to the exteNd Director content repository, the content is stored as a system resource. The repository handles system resources by storing a default set of properties (or metadata) along with content. The following table describes these properties and how default values are assigned:

Property	Default value
Name	Name of file (with extension if provided) **NOTE:** Some WebDAV clients require you to specify extensions for files to indicate the appropriate content editor
Author	Identifier of user who is logged in
Date created	Date uploaded
Abstract	None
Publish date	Null, which means publish as soon as possible
Expiration date	Null, which means never expire

Property	Default value
Checked out by	None

You can change or assign values to these properties in the exteNd Director CM subsystem programmatically or using the CMS Administration Console. Some WebDAV-enabled authoring tools also allow you to edit property values on the client side.

For more information about using the CMS Administration Console, see Chapter 12, "About the CMS Administration Console".

When content is stored as a system resource, it cannot be associated with any custom document types or categories that have been defined in the CM subsystem. To create content that is more tightly integrated with these CM subsystem features, you can:

- Build your own WebDAV client application using a client API provided with the exteNd Director WebDAV subsystem

  For more information about the WebDAV client API, see Chapter 10, "Building Your Own WebDAV Client".
- Use the CM API or the CMS Administration Console to create a document of a particular type in the CM subsystem on the server. You can then edit this content inside a WebDAV-compliant client, preserving the original document type.

How exteNd Director secures content from WebDAV clients

The exteNd Director WebDAV subsystem requires you to provide a valid user ID and password to the WebDAV client. These values are used to authenticate your access privileges when you attempt to access secure content in the content repository from your WebDAV client.

Users do not see resources for which they do not have read access.

For more information, see "Setting up the client" on page 133.

How exteNd Director manages versioning for WebDAV clients

When a WebDAV client requests a resource from the server, the WebDAV subsystem returns the latest version from the content repository—though not necessarily the published version. For example, a WebDAV client cannot retrieve the published version of content if it is not the latest version.

When the WebDAV client uploads and checks in a resource, the WebDAV subsystem creates a new version and publishes it in the content repository.

Installing the exteNd Director WebDAV subsystem

You install the WebDAV subsystem when you create a project in exteNd Director using the EAR Wizard.

- If you have not created an exteNd Director EAR project that includes the WebDAV subsystem, follow the procedure that follows.
- If you have, you are ready to deploy the WebDAV subsystem and can skip to "Deploying the exteNd Director WebDAV subsystem" below the procedure.

➢ **To install the WebDAV subsystem:**

1. Make sure you have installed exteNd Director.
2. Follow the instructions for creating a new exteNd Director EAR project in the chapter on configuring and deploying exteNd Director applications in *Developing exteNd Director Applications*.

   During this process you choose between two setup options: Typical and Custom. If you select Typical setup, the WebDAV subsystem is installed automatically as part of the exteNd Director EAR with the following defaults:

Parameter	Default
Service Context Root	**WebDAVService**
Servlet Path	**main**
Require locks for update operation	disabled

   If you opt for Custom setup, you must include the WebDAV subsystem explicitly and then customize these parameters as needed.
3. After creating the EAR project, select **Archive Layout** and look for **WebDAVService.war**.
4. Expand **WebDAVService.war**, navigate to **WEB-INF/lib**, and double-click **WebDAVService.jar** to view the WebDAV subsystem classes you have just installed in your exteNd Director EAR project.

Now you are ready to deploy the WebDAV subsystem to your J2EE application server.

Deploying the exteNd Director WebDAV subsystem

You deploy the WebDAV subsystem by deploying the EAR in which it resides.

Before you deploy

Before deploying the exteNd Director WebDAV subsystem, you must have installed the following software:

- A J2EE application server
- A WebDAV-enabled client authoring tool

If you are deploying to the Novell exteNd™ Application Server, you must also create a new (empty) database.

📖 For a list of supported application servers and databases, see the exteNd Director *Release Notes*.

Setting up the client

After you deploy the WebDAV subsystem, you can connect a WebDAV-enabled client to the exteNd Director content repository. To establish this connection, you must provide the following parameters to the client:

- User ID and password that are valid for exteNd Director (not a server user ID and password)
- URL that references the directory on the WebDAV server you want to connect to

  The structure of the URL for the Novell WebDAV server is:

  ```
  http://server name/database name/EAR namespace/service context root/servlet path/
  ```

  The structure of the URL for WebLogic or WebSphere WebDAV servers is:

  ```
  http://server name/EAR namespace/service context root/servlet path/
  ```

  For example, if your Novell exteNd Application Server is **localhost**, database name is **Director**, EAR namespace is **DirectorEAR**, service context root is **WebDAVService**, and servlet path is **main**, the URL should look like this:

  ```
  http://localhost/Director/DirectorEAR/WebDAVService/main/
  ```

📖 To learn how to provide these parameters and connect to a site (in this case the exteNd Director content repository) using the WebDAV protocol, consult client documentation.

Supported WebDAV methods

The exteNd Director WebDAV subsystem supports the following WebDAV methods. To learn how to perform these functions from your WebDAV-enabled authoring tool, consult client documentation:

Method	Description
PROPFIND	Retrieves properties on resources and collections from the server. This action is generally transparent to the user; WebDAV client tools use this method to get and display properties such as **name**, **type** (of resource), **date modified**, and **checked out by**.
PROPPATCH	Sets and/or removes properties on server-side resources and collections identified by the Request-URI. This action is generally transparent to the user; WebDAV client tools use this method to modify properties such as **name**, **type** (of resource), **date modified**, and **checked out by**.
COPY	Copies resources and collections on the server—along with their properties—without causing name conflicts. When you copy a collection, all of its members are also copied.
DELETE	Deletes resources or collections on the server.
GET	Retrieves resources and collections from the server, as identified by the Request-URI. Some WebDAV-enabled clients automatically check out resources for you before downloading them from the server; other clients require you to perform two separate operations—first check out the resource, then get it.
HEAD	Functions like GET, but retrieves only header information (without a response message body).
LOCK	Creates a lock specified by the **lockinfo** XML element on the Request-URI. The lockinfo element specifies the scope, type, and owner of the lock. The exteNd Director CM subsystem uses just one type of lock—the exclusive lock, to enforce pessimistic concurrency. The scope of a lock spans the entire state of the resource, including its body and associated properties. Some WebDAV-enabled clients automatically lock resources before you check them out; other clients require you to explicitly lock a resource as a separate operation.

Method	Description
UNLOCK	Removes the lock identified in the Lock-Token request header of the Request-URI. This action unlocks all resources included in the lock. Some WebDAV-enabled clients automatically unlock resources after you check them in; other clients require you to explicitly unlock a resource as a separate operation.
MKCOL	Creates collections on the server.
MOVE	Moves resources and collections on the server without creating name conflicts.
PUT	Uploads resources and collections from the client to the server.
OPTIONS	Returns all methods that can be called on resources and collections specified in the Request-URI. For example, if the resource is a document, OPTIONS returns LOCK, UNLOCK, OPTIONS, GET, PUT, MOVE, DELETE, COPY, PROPFIND, and PROPPATCH.

Public WebDAV server

Novell provides a WebDAV server—deployed and publicly available—against which you can test your WebDAV clients. This server provides the features of the Novell WebDAV implementation.

CAUTION: *Do not use this server for production applications. Novell cannot be responsible for content uploaded by anonymous users, and periodically purges user data.*

➢ **To access the Novell public WebDAV server (general steps):**

1. Access the server from your WebDAV client using this URL:

   `http://webdav.silverstream.com/Director/WebDAVService/main`

2. When prompted, provide these credentials:

Credential	Value
User ID	**devcenter**
Password	**rocks**

➢ **To access the Novell public WebDAV server from a Windows 2000 SP2 client:**

1. Start **My Network Places**.
2. Double-click **Add a Network Place**.

   The Add Network Place Wizard opens.
3. Enter the URL for the public WebDAV server:

   ```
   http://webdav.silverstream.com/Director/WebDAVService/main
   ```

   The wizard prompts you for credentials.
4. When prompted, provide these credentials:

Credential	Value
User name	devcenter
Password	rocks

5. Click **OK**.

   The wizard prompts you for a server connection name.
6. Enter a name—for example, **Public WebDAV Server**—and click **Finish**.

   The connection to the public WebDAV server is established.

10 Building Your Own WebDAV Client

This chapter describes an API provided with the exteNd Director WebDAV service for developing a custom WebDAV client that takes advantage of the specialized features of the exteNd Director Content Management (CM) subsystem to create and administer content.

The chapter covers the following topics:

- About the WebDAV client API
- Why build your own WebDAV client?
- Configuring your environment
- Using the WebDAV client API
- Programming practices using helper methods
- Programming practices using utility methods
- Issuing WebDAV requests from a Java client

About the WebDAV client API

The WebDAV client API is based on the Jakarta Slide content management framework and is designed to work with the exteNd Director CM subsystem. Slide is a low-level framework that can be used to develop a consistent interface for manipulating binary content in a variety of data stores using the WebDAV protocol.

Java client applications can access the Slide content management framework directly through a set of Java classes that implement WebDAV methods and other low-level logic in these functional areas:

- Managing the namespace (for creating, moving, copying, and deleting content)
- Updating content and metadata

- Locking and unlocking content
- Securing content

The exteNd Director WebDAV client API adds a level of abstraction by providing wrapper classes around the Slide client API. These classes contain helper and utility methods that encapsulate the low-level Slide methods and add logic that tightly integrates with the specialized capabilities of the exteNd Director CM subsystem. For example, you can build a WebDAV client that assigns categories to documents, associates custom metadata with content, and creates content using custom templates called *document types* as defined in the CM subsystem.

 For more information about Slide, see the Jakarta Slide project Web site. The following URL was valid at the time this chapter was published:

`http://jakarta.apache.org/slide/`

Why build your own WebDAV client?

With so many commercial and open source WebDAV client applications now available—and more on the way—why build your own WebDAV client to work with the exteNd Director CM subsystem?

Here is a key reason: to tailor an application to your unique authoring needs in terms of creating, updating, and managing content using the exteNd Director CM subsystem. With this objective in mind, the WebDAV client API allows you to develop applications that are more robust than most commercial and open-source WebDAV clients, because it provides:

- Simplified access to all WebDAV methods, including PROPPATCH
- An interface to the comprehensive content management features of the CM subsystem, including the ability to create documents using custom templates and manipulate custom metadata separately from content

Configuring your environment

To **use the WebDAV client API**, you must add the following JAR files to your project classpath:

JAR file	Description
WebDAV_slide.jar	Contains relevant Slide client API classes
WebDAVClient.jar	Contains exteNd Director WebDAV client API classes

These JAR files are installed with exteNd Director in the following location in the exteNd Director installation directory:

```
eXtendDirector\utilities\Client
```

To **run a WebDAV client**, you must add the following JAR files to your client's classpath at runtime:

JAR file	Directory	Location
WebDAVClient.jar	exteNd Director installation directory	\utilities\Client
WebDAV_slide.jar		\utilities\Client
xerces.jar		\lib
xalan.jar		\lib
FrameworkService.jar		\lib
servlet.jar	Novell installation directory	\lib

For example: to run a WebDAV Java client program called getDocuments, enter these commands—substituting your own installation directory paths:

```
>set classpath=D:\Director_install\eXtendDirector\utilities\Client\WebDAVClient.jar;D:\
Director_install\eXtendDirector\utilities\Client\WebDAV_slide.jar;.;D:\Director_install\
eXtendDirector\lib\xerces.jar;D:\Director_install\eXtendDirector\lib\xalan.jar;D:\
Director_install\eXtendDirector\lib\FrameworkService.jar;d:\Director_install\eXtendServer\
lib\servlet.jar

>java -cp %classpath% getDocuments
```

Using the WebDAV client API

You use the WebDAV client API to design a custom authoring tool with WebDAV access to the exteNd Director CM subsystem for managing collaborative interactions with your content.

You need to build your own user interface, but the API provides the logical underpinnings for invoking key CM functions from your client:

- Creating documents using custom templates
- Categorizing documents
- Deleting, copying, moving, and renaming resources and collections
- Locking and unlocking documents
- Making collections
- Updating documents
- Getting and setting custom metadata values in a document

WebDAV requests and responses

The WebDAV client API provides methods that invoke CM functions by sending WebDAV requests. The result of each request is returned as a WebDAV response that includes a status code to indicate success or the reason for failure.

A WebDAV request consists of a header and a body. The request header contains the method, target resource, HTTP version, and a sequence of key/value pairs containing parameters for the method. The request body defines additional—and perhaps more complex—parameters if necessary.

Similarly, a WebDAV response contains a header and optional body. The response header contains information about the response, such as the HTTP version used by the server, along with status codes and messages. The response body generally contains the result of method execution—such as a document.

Classes in the WebDAV client API provide methods for easily constructing and sending specific WebDAV requests and processing responses.

📖 For more information about WebDAV, search on the Web for **rfc2518**—the WebDAV specification.

Working with resources, collections, and properties

WebDAV requests act on Web resources, collections, and properties as described in "Information elements for distributed Web authoring" on page 128. When you issue a WebDAV request, you need to pass along a reference to the element of interest. This reference should be a URI, relative to the element's server in this format:

```
/database name/WebDAVService/main/path relative to default (root) folder
```

For example, assume your exteNd Director database is called **Director**. For a document called **MyDocument** that resides in a folder called **Test** in the default (root) folder, the URI looks like this:

```
/Director/WebDAVService/main/Test/MyDocument
```

Classes

The WebDAV client API consists of these key classes:

Class	Description
EboDAVSwitch	Constructs WebDAV requests and fetches WebDAV responses
EboDAVException	Defines WebDAV exceptions
EboDAVStatus	Indicates the status code associated with WebDAV exceptions

EboDAVSwitch—the heart of the matter The EboDAVSwitch object is the heart of the WebDAV client API, containing most of the functionality for communicating with the CM subsystem. EboDAVSwitch provides helper methods and utility methods that encapsulate much of the low-level Slide code required for transmitting WebDAV requests and responses.

Helper methods

The EboDAVSwitch object provides a set of helper methods for constructing WebDAV requests. Each helper method allows you to send a complete request in a single line of code.

Here is list of supported WebDAV requests that have associated helper methods. Click on the links in the table to get more information about how to code specific WebDAV requests in a Java client program.

Request	Helper method
Adding a category reference to a document	addCategoryToDocument
Deleting a document	copyElement
Creating a new collection	makeCollection
Creating a new document from a custom template	createNewDocument
Deleting a document	deleteDocument
Locking a document	lockDocument
Moving a resource or collection	moveElement
Removing a category reference from a document	removeCategoryFromDocument

Request	Helper method
Removing all category references from a document	removeAllCategoriesFrom Document
Renaming a resource or collection	moveElement
Setting the value of a custom field in a document	setFieldValueForDocument
Unlocking a document	unlockDocument
Updating a document	putDocument

Some WebDAV requests do not have associated helper methods and can be issued only by using Slide classes and utility methods, described next.

 For information on how to use these helper methods in WebDAV client applications, see "Programming practices using helper methods" on page 144.

Utility methods

All WebDAV requests can be invoked using utility methods. Compared to helper methods, utility methods expose more of the Slide API than helper methods. The tradeoff is that while you gain access to the additional functionality offered by the Slide API, you'll have to write more lines of code to send a WebDAV request.

Utility methods also provide a mechanism for issuing WebDAV requests that do not have associated helper methods.

Utility methods that wrap Slide functions

Here is list of commonly used utility methods that wrap Slide functions for constructing and issuing WebDAV requests:

Utility method	What it does
createCredentials	Creates credentials **NOTE:** The *credentials* object is a Slide object that is used for authenticating users
createWebDAVmethod	Creates the method you want to execute as part of your WebDAV request
endSession	Ends a WebDAV client session
executeCommand	Issues the WebDAV request

Utility method	What it does
getState	Gets state **NOTE:** The *state* object is a Slide object; you call the Slide method setAuthenticateToken on the state object to indicate how you are going to authenticate users
setCredentials	Sets credentials on the EboDAVSwitch object
setState	Sets state with your authentication token
startSession	Starts a WebDAV client session

 For information about how to use these utility methods in WebDAV client applications, see "Programming practices using utility methods" on page 146.

Associated Slide API classes

When you work with utility methods, you need to use several Slide API classes:

- `Credentials`
- `State`
- Specific WebDAV method classes:
  - `CopyMethod`
  - `DeleteMethod`
  - `GetMethod`
  - `HeadMethod`
  - `LockMethod`
  - `MoveMethod`
  - `OptionsMethod`
  - `PostMethod`
  - `PropFindMethod`
  - `PropPatchMethod`
  - `PutMethod`
  - `UnlockMethod`

 For more information about these classes, see the Slide WebDAV client JavaDoc, available at this URL (valid at the time this chapter was published):

`http://jakarta.apache.org/slide/clientjavadoc/index.html`

WebDAV requests that have no helper methods

Here is list of WebDAV requests that have no associated exteNd Director helper methods and therefore can be implemented only by using Slide classes and exteNd Director utility methods. Click on the links in the table to get more information about how to code these WebDAV requests in a Java client program:

WebDAV request	Associated WebDAV method
Getting a resource or collection	GET
Getting header information from a resource or collection	HEAD
Getting methods that can be called on a resource or collection	OPTIONS
Getting properties defined on a resource or collection	PROPFIND

Programming practices

This section describes best practices for using the client API to issue WebDAV requests and process WebDAV responses in custom Java client programs. The logic varies depending on whether you use helper methods or utility methods.

Programming practices using helper methods

Recommended steps

Here are the steps for using helper methods to issue WebDAV requests:

1. Instantiate an EboDAVSwitch object.
2. Start a session on the EboDAVSwitch object.
3. Call the helper method on the EboDAVSwitch object in a try/catch block.
4. Get the response and process the results if necessary.
5. End the session.

Code example: deleting a document using a helper method

Here is sample code showing how to use the helper method **deleteDocument()** in a WebDAV client. In this example, assume server URL = **localhost** and port = **80**. The document to be deleted is passed as an argument to the method.

Note that an EboDAVStatus object is also instantiated. This object is used to check the status of the request and inform the user of success or failure.

```
/**
    deleteADocument

*/
import com.sssw.webdav.client.*;

public class deleteADocument {

    private static boolean m_debug = false;

    public void deleteADocument (String document)
    {

        //Define variables
        int statuscode = 0;
        String user = "contentadmin";
        String password = "contentadmin";
        String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";

        //Instantiate an EboDAVSwitch object
        EboDAVSwitch dav = new EboDAVSwitch();

        //Instantiate an EboDAVStatus object
        EboDAVStatus status = new EboDAVStatus();

        //Start a session
        dav.startSession("localhost", 80);


        try
       {
            //Lock document before trying to delete
            statuscode = dav.lockDocument(user, password, realm, document);
            if (statuscode==EboDAVStatus.SC_NO_CONTENT)
               System.out.println("Request succeeded: The document is now locked");
            else
               System.out.println("Request failed: " + status.getStatusText(statuscode));

            //Send the WebDAV request to delete document
            statuscode = dav.deleteDocument(user, password, realm, document);
            if (statuscode==EboDAVStatus.SC_OK)
               System.out.println("Request succeeded: The document was deleted.");
            else
               System.out.println("Request failed: " + status.getStatusText(statuscode));

       }
        catch (EboDAVException e) {
            if (m_debug)
               e.printStackTrace();
            else
               System.out.println(e.getMessage());
```

```
        }

         //End session
         dav.endSession();
    }
}
```

To learn how to issue the same WebDAV request using utility methods, see "Code example: deleting a document using utility methods" on page 147.

Programming practices using utility methods

Recommended steps

Here are the steps for using utility methods to issue WebDAV requests:

1. Instantiate an EboDAVSwitch object.
2. Start a session on the EboDAVSwitch object.
3. Create and set credentials on the EboDAVSwitch object.
4. Get and set the state of the EboDAVSwitch object and the authentication realm.
5. Construct the WebDAV method.
6. Execute the WebDAV method.
7. End the session.

Constructing WebDAV requests that use Proppatch

The WebDAV Proppatch method is used with exteNd Director utility methods to issue a variety of WebDAV requests:

- Adding a category reference to a document
- Removing a category reference from a document
- Removing all category references from a document
- Setting the value of a custom field in a document

For each of these requests, you must instantiate a Slide PropPatchMethod object, then call the addPropertyToSet() method on the PropPatchMethod object using this signature:

```
addPropertyToSet( String property name, String property value,
String namespace-abbr, String namespace )
```

Here are descriptions of the arguments to addPropertyToSet():

Argument	Description
property name	The name or UUID of the property to be updated
property value	The value of the property to be updated If *property name* is a UUID, then *property value* must be null
namespace-abbr	An arbitrary string that must be unique within the PropPatch method request
namespace	Type of request issued using the PropPatch method These requests are defined as fields of the EboWebdavConstants class: ◆ EboWebdavConstants.PROPPATCH_SETFIELDVALUE ◆ EboWebdavConstants.PROPPATCH_ADDCATEGORY ◆ EboWebdavConstants.PROPPATCH_REMOVECATEGORY ◆ EboWebdavConstants.PROPPATCH_REMOVEALL CATEGORIES

Setting values of standard fields You can also use the WebDAV Proppatch method to set values of standard fields—such as title and author—in a document. In this case, call addPropertyToSet() without the **namespace-abbr** and **namespace** arguments.

Code example: deleting a document using utility methods

Here is sample code illustrating how to use utility methods with Slide classes in a WebDAV client to send a request to delete a document. In this example, assume server URL = **localhost** and port = **80**. The example uses the following Slide classes:

- Credentials
- State
- DeleteMethod

```
/**
    deleteTheDocument

*/
import com.sssw.webdav.client.*;
import org.apache.webdav.lib.*;
import org.apache.webdav.lib.methods.*;

public class deleteTheDocument
{
   private static boolean m_debug = false;
```

```
   public void deleteTheDocument (String document)
   {
      //Define variables
      int statuscode = 0;
      String user = "contentadmin";
      String password = "contentadmin";
      String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";

      //Instantiate an EboDAVSwitch object
      EboDAVSwitch dav = new EboDAVSwitch();

      //Instantiate an EboDAVStatus object
      EboDAVStatus status = new EboDAVStatus();

      //Start a session
      dav.startSession("localhost", 80);

      //Get and set credentials
      Credentials credentials = dav.createCredentials(user, password);
      dav.setCredentials(credentials);

      //Get and set state and authentication realm
      State state = dav.getState();
      state.setAuthenticateToken(realm);
      dav.setState(state);


      try
      {
          //Create the WebDAV method object LockMethod
          LockMethod lm = (LockMethod)dav.createWebdavMethod(dav.LOCK_METHOD,document);

          //Set the owner
          lm.setOwner(user);

          //Execute LockMethod
          dav.executeCommand(lm);
          statuscode = lm.getStatusCode();
          if (statuscode == (EboDAVStatus.SC_NO_CONTENT))
             System.out.println("Request succeeded: The document was locked.");
          else
             System.out.println("Request failed: " + status.getStatusText(statuscode));

          //Create the WebDAV method object DeleteMethod
          DeleteMethod dm =
(DeleteMethod)dav.createWebdavMethod(dav.DELETE_METHOD,document);

          //Execute DeleteMethod (send the WebDAV request to delete document)
          dav.executeCommand(dm);
          statuscode = dm.getStatusCode();
          if (statuscode == (EboDAVStatus.SC_OK))
             System.out.println("Request succeeded: The document was deleted.");
          else
             System.out.println("Request failed: " + status.getStatusText(statuscode));
```

```
        }
        catch (EboDAVException e)
        {
           if (m_debug)
              e.printStackTrace();
           else
              System.out.println(e.getMessage());
        }
        catch (java.net.MalformedURLException murle)
        {
           if (m_debug)
              murle.printStackTrace();
           else
              System.out.println(murle.getMessage());
        }
        catch (java.io.IOException ioe)
        {
          if (m_debug)
            ioe.printStackTrace();
          else
            System.out.println(ioe.getMessage());
        }

        //End session
        dav.endSession();
    }
}
```

To learn how to issue the same WebDAV request using helper methods, see "Code example: deleting a document using a helper method" on page 144.

Issuing WebDAV requests from a Java client

This section describes how to issue WebDAV requests from a Java client application. The following functions are covered:

- Adding a category reference to a document
- Copying a resource or collection
- Creating a new collection
- Creating a new document from a custom template
- Deleting a document
- Getting a resource or collection
- Getting header information from a resource or collection
- Getting methods that can be called on a resource or collection
- Getting properties defined on a resource or collection
- Locking a document

- Moving a resource or collection
- Removing a category reference from a document
- Removing all category references from a document
- Renaming a resource or collection
- Setting the value of a custom field in a document
- Unlocking a document
- Updating a document

Adding a category reference to a document

The following code examples show how to add a category reference to a document. A category is a descriptive name used to group documents logically in the CM subsystem.

Code example: adding a category reference using a helper method

This example uses the helper method **addCategoryToDocument()**:

```
/**
    addCategoryReferenceToDocument

*/
import com.sssw.webdav.client.*;

public class addCategoryReferenceToDocument {

    private static boolean m_debug = false;

    public void addCategoryReferenceToDocument (String document, String categoryUUID)
    {

      //Define variables
      int statuscode = 0;
      String user = "contentadmin";
      String password = "contentadmin";
      String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";

      //Instantiate an EboDAVSwitch object
      EboDAVSwitch dav = new EboDAVSwitch();

      //Instantiate an EboDAVStatus object
      EboDAVStatus status = new EboDAVStatus();

      //Start a session
      dav.startSession("localhost", 80);


      try
      {
```

```
         //Lock the document
         statuscode = dav.lockDocument(user, password, realm, document);
         if (statuscode == EboDAVStatus.SC_NO_CONTENT)
            System.out.println("Request succeeded: The category was added to " +
document);
         else
            System.out.println("Request failed: " + status.getStatusText(statuscode));

         //Send the WebDAV request to add a category reference
         statuscode = dav.addCategoryToDocument(user, password, realm, document,
categoryUUID);
         if (statuscode==EboDAVStatus.SC_MULTI_STATUS)
            System.out.println("Request succeeded: The category was added to " +
document);
         else
            System.out.println("Request failed: " + status.getStatusText(statuscode));

         //Unlock the document
         statuscode = dav.unlockDocument(user, password, realm, document);
         if (statuscode == EboDAVStatus.SC_NO_CONTENT)
            System.out.println("Request succeeded: The document was unlocked.");
         else
            System.out.println("Request failed: " + status.getStatusText(statuscode));
      }
      catch (EboDAVException e)
      {
         if (m_debug)
            e.printStackTrace();
         else
            System.out.println(e.getMessage());
      }

      //End session
      dav.endSession();
   }
}
```

Code example: adding a category reference using utility methods

This example uses the Slide **PropPatchMethod** class and the exteNd Director utility methods **startSession()**, **createCredentials()**, **setCredentials()**, **getState()**, **setState()**, and **createWebDAVMethod()**.

The method that adds the category reference is addPropertyToSet(), called on the PropPatchMethod object. Notice that the second argument—*property value*—is null (because the category UUID is passed as the first argument—*property name)*. For more information about addPropertyToSet() and its arguments, see "Constructing WebDAV requests that use Proppatch" on page 146.

```
/**
    addCategoryReference

*/
import com.sssw.webdav.client.*;
import com.sssw.webdav.common.EboWebdavConstants;
import org.apache.webdav.lib.*;
import org.apache.webdav.lib.methods.*;

public class addCategoryReference
{
   private static boolean m_debug = false;

   public void addCategoryReference (String document, String categoryUUID)
   {
      //Define variables
      int statuscode = 0;
      String user = "contentadmin";
      String password = "contentadmin";
      String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";
      String namespace-abbr = "AC";

      //Instantiate an EboDAVSwitch object
      EboDAVSwitch dav = new EboDAVSwitch();

      //Instantiate an EboDAVStatus object
      EboDAVStatus status = new EboDAVStatus();

      //Start a session
      dav.startSession("localhost", 80);

      //Get and set credentials
      Credentials credentials = dav.createCredentials(user, password);
      dav.setCredentials(credentials);

      //Get and set state and authentication realm
      State state = dav.getState();
      state.setAuthenticateToken(realm);
      dav.setState(state);

      try
      {

         //Lock the document
         //Create the WebDAV method object LockMethod
         LockMethod lm = (LockMethod)dav.createWebdavMethod(dav.LOCK_METHOD,document);

         //Set the owner
         lm.setOwner(user);

         //Execute the command
         dav.executeCommand(lm)
         statuscode = lm.getStatusCode();
         if (statuscode == (EboDAVStatus.SC_NO_CONTENT))
```

```
            System.out.println("Request succeeded: The document was locked.");
         else
            System.out.println("Request failed: " + status.getStatusText(statuscode));

         //Create the WebDAV method object PropPatchMethod
         PropPatchMethod ppm =
(PropPatchMethod)dav.createWebdavMethod(dav.PROPPATCH_METHOD,document);
         ppm.addPropertyToSet( categoryUUID, null, namespace-abbr,
EboWebdavConstants.PROPPATCH_ADDCATEGORY);

         //Execute PropPatchMethod (send the WebDAV request to add category reference)
         dav.executeCommand(ppm);
         statuscode = ppm.getStatusCode();
         if (statuscode == (EboDAVStatus.SC_MULTI_STATUS))
            System.out.println("Request succeeded: The category was added to " + document
+ ".");
         else
            System.out.println("Request failed: " + status.getStatusText(statuscode));

         //Create the WebDAV method object UnlockMethod
         UnlockMethod ulm =
(UnlockMethod)dav.createWebdavMethod(dav.UNLOCK_METHOD,document);

         //Execute UnlockMethod
         dav.executeCommand(ulm);
         statuscode = ulm.getStatusCode();
         if (statuscode == (EboDAVStatus.SC_NO_CONTENT))
            System.out.println("Request succeeded: The document was unlocked.");
         else
            System.out.println("Request failed: " + status.getStatusText(statuscode));
      }
      catch (EboDAVException e)
      {
         if (m_debug)
            e.printStackTrace();
         else
            System.out.println(e.getMessage());
      }
      catch (java.net.MalformedURLException murle)
      {
         if (m_debug)
            murle.printStackTrace();
         else
            System.out.println(murle.getMessage());
      }
      catch (java.io.IOException ioe)
      {
         if (m_debug)
            ioe.printStackTrace();
         else
            System.out.println(ioe.getMessage());
      }
```

```
      //End session
      dav.endSession();
   }
}
```

Copying a resource or collection

The following code shows how to copy a document from a source path to a destination path. In this case the source path points to a document. To copy other types of resources or collections, make sure the source path points to the element of interest.

Code example: copying a document using a helper method

The example uses the helper method **copyElement()**:

```
/**
    copyADocument

*/
import com.sssw.webdav.client.*;

public class copyADocument {

    private static boolean m_debug = false;

    public void copyADocument (String docsourcepath, String docdestinationpath)
    {

       //Define variables
       int statuscode = 0;
       String user = "contentadmin";
       String password = "contentadmin";
       String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";
       boolean overwrite = true; //Overwrite an existing document of the same name in the
docdestinationpath
       boolean autogen = true; //Generate folders in the docdestinationpath if they don't
exist

       //Instantiate an EboDAVSwitch object
       EboDAVSwitch dav = new EboDAVSwitch();

       //Instantiate an EboDAVStatus object
       EboDAVStatus status = new EboDAVStatus();

       //Start a session
       dav.startSession("localhost", 80);

       //Send the WebDAV request to copy document
       try
       {
```

```
          statuscode = dav.copyElement(user, password, realm, docsourcepath,
docdestinationpath, overwrite, autogen);
          if (statuscode==EboDAVStatus.SC_CREATED)
             System.out.println("Request succeeded: The document " + docsourcepath + "was
copied to " + docdestinationpath);
          else
             System.out.println("Request failed: " + status.getStatusText(statuscode));
       }
       catch (EboDAVException e)
       {
          if (m_debug)
             e.printStackTrace();
          else
             System.out.println(e.getMessage());
       }

       //End session
       dav.endSession();
   }
}
```

 You can also copy a resource or collection using the Slide **CopyMethod** class and exteNd Director utility methods. See "Programming practices using utility methods" on page 146.

Creating a new collection

The following code shows how to create a new *collection*. Recall that a collection is a container for other resources and collections. A folder is a an example of a collection.

Code example: creating a collection using a helper method

This example uses the helper method **makeCollection()**:

```
/**
    makeACollection

*/
import com.sssw.webdav.client.*;

public class makeACollection {

    private static boolean m_debug = false;

    public void makeACollection (String parent_folder, String folder_name)
    {

   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
```

```
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();

   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);

   //Send the WebDAV request to make a collection
   try
   {
      statuscode = dav.makeCollection(user, password, realm, parent_folder, folder_name,
true);
      if (statuscode==EboDAVStatus.SC_CREATED)
         System.out.println("Request succeeded: The collection " + parent_folder + "/" +
folder_name + "was created.");
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }

   //End session
   dav.endSession();
   }
}
```

You can also make a new collection using the Slide **MkcolMethod** class and exteNd Director utility methods. See "Programming practices using utility methods" on page 146.

Creating a new document from a custom template

The following code shows how to create a new document from a custom template. Custom templates are document types that you define in the exteNd Director CM subsystem using the CM API or CMS Administration Console.

The document that is created contains the content "Hello world!" along with any custom fields defined in the document type.

Code example: creating a document using a helper method

This example uses the helper method **createNewDocument()**. The document type is passed as an argument to createNewDocument, along with the user name, password, realm, containing folder, and content:

```
/**
    createADocument

*/
import com.sssw.webdav.client.*;

public class createADocument {

    private static boolean m_debug = false;

    public void createADocument (String document, String folder, String documentType)
    {

       //Define variables
       int statuscode = 0;
       String user = "contentadmin";
       String password = "contentadmin";
       String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";
       String sourcetext = "Hello world!";
       byte [] content = sourcetext.getBytes();

       //Instantiate an EboDAVSwitch object
       EboDAVSwitch dav = new EboDAVSwitch();

       //Instantiate an EboDAVStatus object
       EboDAVStatus status = new EboDAVStatus();

       //Start a session
       dav.startSession("localhost", 80);

       //Send the WebDAV request to create a document
       try
       {
          statuscode = dav.createNewDocument(user, password, realm, folder, document,
documentType, content);
          if (statuscode==EboDAVStatus.SC_CREATED)
             System.out.println("Request succeeded: The document " + document + "was
created.");
          else
             System.out.println("Request failed: " + status.getStatusText(statuscode));

       }
       catch (EboDAVException e)
       {
          if (m_debug)
             e.printStackTrace();
          else
             System.out.println(e.getMessage());
```

```
        }

        //End session
        dav.endSession();
    }
}
```

Deleting a document

For examples of how to delete a document from a WebDAV client, see "Code example: deleting a document using a helper method" on page 144 and "Code example: deleting a document using utility methods" on page 147.

Getting a resource or collection

The following code shows how to get the content of a document stored in the CM subsystem. The document is referenced as the second argument of the createWebDAVMethod() utility method. To get other types of resources or collections, modify this argument to point to the element of interest.

Code example: getting a document using utility methods

This example uses the Slide **GetMethod** class and the exteNd Director utility methods **startSession()**, **createCredentials()**, **setCredentials()**, **getState()**, **setState()**, and **createWebDAVMethod()**.

By calling the getDataAsString() method on the GetMethod class, the client application retrieves the content of the document in HTML format.

There is no helper method for getting a resource or collection:

```
/**
    getTheDocument

*/
import com.sssw.webdav.client.*;
import org.apache.webdav.lib.*;
import org.apache.webdav.lib.methods.*;

public class getTheDocument
{
   private static boolean m_debug = false;

   public void getTheDocument (String document)
   {
   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
```

```
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();

   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);

   //Get and set credentials
   Credentials credentials = dav.createCredentials(user, password);
   dav.setCredentials(credentials);

   //Get and set state and authentication realm
   State state = dav.getState();
   state.setAuthenticateToken(realm);
   dav.setState(state);

   //Create the WebDAV method object GetMethod
   GetMethod gm = (GetMethod)dav.createWebdavMethod(dav.GET_METHOD,document);

   //Execute GetMethod (send the WebDAV request to get document)
   try
   {
      dav.executeCommand(gm);
      statuscode = gm.getStatusCode();
      if (statuscode == (EboDAVStatus.SC_OK))
      {
         String html = gm.getDataAsString();
         System.out.println("Request succeeded: Got the document and its content as
html.");
      }
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }
   catch (java.net.MalformedURLException murle)
   {
      if (m_debug)
         murle.printStackTrace();
      else
         System.out.println(murle.getMessage());
   }
   catch (java.io.IOException ioe)
   {
```

```
      if (m_debug)
         ioe.printStackTrace();
      else
         System.out.println(ioe.getMessage());
   }


   //End session
   dav.endSession();
   }
}
```

There is no exteNd Director helper method for getting a resource or collection.

Getting header information from a resource or collection

The following code shows how to get the header information of a document stored in the CM subsystem. The document is referenced as the second argument of the createWebDAVMethod() utility method. To get other types of resources or collections, modify this argument to point to the element of interest.

Code example: getting header information using utility methods

This example uses the Slide **HeadMethod** class and the exteNd Director utility methods **startSession()**, **createCredentials()**, **setCredentials()**, **getState()**, **setState()**, and **createWebDAVMethod()**.

There is no helper method for getting a resource or collection:

```
/**
    getDocumentHeader

*/
import com.sssw.webdav.client.*;
import org.apache.webdav.lib.*;
import org.apache.webdav.lib.methods.*;

public class getDocumentHeader
{
   private static boolean m_debug = false;

   public void getDocumentHeader (String document)
   {
   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";
   String authtype = "";

   //Instantiate an EboDAVSwitch object
```

```
   EboDAVSwitch dav = new EboDAVSwitch();

   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);

   //Get and set credentials
   Credentials credentials = dav.createCredentials(user, password);
   dav.setCredentials(credentials);

   //Get and set state and authentication realm
   State state = dav.getState();
   state.setAuthenticateToken(realm);
   dav.setState(state);

   //Create the WebDAV method object HeadMethod
   HeadMethod hm = (HeadMethod)dav.createWebdavMethod(dav.HEAD_METHOD,document);

   //Execute HeadMethod (send the WebDAV request to get document header)
   try
   {
      dav.executeCommand(hm);
      statuscode = hm.getStatusCode();
      if (statuscode == (EboDAVStatus.SC_OK))
      {
         //Get authorization type from header
         authtype = hm.getHeader ("authorization").toString();
         System.out.println("Request succeeded: Got the document header. Authorization
type is " + authtype);
      }
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }
   catch (java.net.MalformedURLException murle)
   {
      if (m_debug)
         murle.printStackTrace();
      else
         System.out.println(murle.getMessage());
   }
   catch (java.io.IOException ioe)
   {
      if (m_debug)
         ioe.printStackTrace();
```

```
      else
         System.out.println(ioe.getMessage());
   }


   //End session
   dav.endSession();
   }
}
```

There is no exteNd Director helper method for getting header information from a resource or collection.

Getting methods that can be called on a resource or collection

The following code shows how to get the methods that can be called on a document stored in the CM subsystem. The document is referenced as the second argument of the createWebDAVMethod() utility method. To get other types of resources or collections, modify this argument to point to the element of interest.

Code example: getting allowed methods using utility methods

This example uses the Slide **OptionsMethod** class and the exteNd Director utility methods **startSession()**, **createCredentials()**, **setCredentials()**, **getState()**, **setState()**, and **createWebDAVMethod()**.

There is no helper method for getting allowed methods on a resource or collection:

```
/**
    getAllowedMethods

*/
import com.sssw.webdav.client.*;
import org.apache.webdav.lib.*;
import org.apache.webdav.lib.methods.*;
import java.util.*;

public class getAllowedMethods
{
   private static boolean m_debug = false;

   public void getAllowedMethods (String document)
   {
   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();
```

```
//Instantiate an EboDAVStatus object
EboDAVStatus status = new EboDAVStatus();

//Start a session
dav.startSession("localhost", 80);

//Get and set credentials
Credentials credentials = dav.createCredentials(user, password);
dav.setCredentials(credentials);

//Get and set state and authentication realm
State state = dav.getState();
state.setAuthenticateToken(realm);
dav.setState(state);

//Create the WebDAV method object HeadMethod
OptionsMethod om = (OptionsMethod)dav.createWebdavMethod(dav.OPTIONS,document);

//Execute OptionsMethod (send the WebDAV request to get the allowed methods on
 //the document)
try
{
   dav.executeCommand(om);
   statuscode = om.getStatusCode();
   if (statuscode == (EboDAVStatus.SC_OK))
   {
      System.out.println("Request succeeded: Got the document header.\n");
      System.out.println("The allowed methods on " + document + " are:\n");
      Enumeration methods = om.getAllowedMethods();
      while (methods.hasMoreElements()) {
         System.out.println( methods.nextElement().toString() + "\n" );
      }
   }
   else
      System.out.println("Request failed: " + status.getStatusText(statuscode));

}
catch (EboDAVException e)
{
   if (m_debug)
      e.printStackTrace();
   else
      System.out.println(e.getMessage());
}
catch (java.net.MalformedURLException murle)
{
   if (m_debug)
      murle.printStackTrace();
   else
      System.out.println(murle.getMessage());
}
catch (java.io.IOException ioe)
{
```

```
      if (m_debug)
         ioe.printStackTrace();
      else
         System.out.println(ioe.getMessage());
   }


   //End session
   dav.endSession();
   }
}
```

There is no exteNd Director helper method for getting methods that can be called on a resource or collection.

Getting properties defined on a resource or collection

The following code shows how to get properties defined on a document stored in the CM subsystem. The document is referenced as the second argument of the createWebDAVMethod() utility method. To get other types of resources or collections, modify this argument to point to the element of interest.

Code example: getting properties using utility methods

This example uses the Slide **PropFindMethod** class and the exteNd Director utility methods **startSession()**, **createCredentials()**, **setCredentials()**, **getState()**, **setState()**, and **createWebDAVMethod()**.

There is no helper method for getting properties defined on a resource or collection:

```
/**
    getProperties

*/
import com.sssw.webdav.client.*;
import org.apache.webdav.lib.*;
import org.apache.webdav.lib.methods.*;
import java.util.*;

public class getProperties
{
   private static boolean m_debug = false;

   public void getProperties (String document)
   {
   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";
```

```
   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();

   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);

   //Get and set credentials
   Credentials credentials = dav.createCredentials(user, password);
   dav.setCredentials(credentials);

   //Get and set state and authentication realm
   State state = dav.getState();
   state.setAuthenticateToken(realm);
   dav.setState(state);

   //Create the WebDAV method object PropFindMethod
   PropFindMethod pfm =
(PropFindMethod)dav.createWebdavMethod(dav.PROPFIND_METHOD,document);

   //Execute PropFindMethod (send the WebDAV request to get the properties defined on
    //the document)
   try
   {
      dav.executeCommand(pfm);
      statuscode = pfm.getStatusCode();
      if (statuscode == (EboDAVStatus.SC_MULTI_STATUS))
      {
         System.out.println("Request succeeded: Got the properties.\n");
         System.out.println("The properties defined on " + document + " are:\n");
         Enumeration props = pfm.getResponseProperties(document);
         while (props.hasMoreElements()) {
            System.out.println( props.nextElement().toString() + "\n" );
         }
      }
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }
   catch (java.net.MalformedURLException murle)
   {
      if (m_debug)
         murle.printStackTrace();
      else
         System.out.println(murle.getMessage());
```

```
   }
   catch (java.io.IOException ioe)
   {
      if (m_debug)
         ioe.printStackTrace();
      else
         System.out.println(ioe.getMessage());
   }


   //End session
   dav.endSession();
   }
}
```

There is no exteNd Director helper method for getting methods that can be called on a resource or collection.

Locking a document

The following code shows how to lock a document for exclusive access in a collaborative environment. You might invoke this function in your WebDAV client application when a user checks out a document.

Code example: locking a document using a helper method

The example uses the helper method **lockDocument()**. This method throws an exception if the document of interest is already locked. To explicitly check for this condition, the code calls the checkLockToken() method:

```
/**
    lockADocument

*/
import com.sssw.webdav.client.*;

public class lockADocument {

    private static boolean m_debug = false;

    public void lockADocument (String document)
    {

   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();
```

```
   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);
   try
      {
       //If document not already locked, send the WebDAV request to lock the document
       if ( dav.checkLockToken(document) == null)
       {
           statuscode = dav.lockDocument(user, password, realm, document);
           if (statuscode==EboDAVStatus.SC_NO_CONTENT)
              System.out.println("Request succeeded: The document " + document + "was
locked.");
           else
              System.out.println("Request failed: " + status.getStatusText(statuscode));
       }
        else
           System.out.println("Document is already locked.");

      }
      catch (EboDAVException e)
      {
         if (m_debug)
            e.printStackTrace();
         else
            System.out.println(e.getMessage());
      }

   //End session
   dav.endSession();
   }
}
```

You can also lock a document using the Slide **LockMethod** class and exteNd Director utility methods. See "Programming practices using utility methods" on page 146.

Moving a resource or collection

The following code shows how to move a folder from a source path to a destination path. In this case, the source path points to a folder. To move other types of resources or collections, make sure the source path points to the element of interest.

Code example: moving a folder using a helper method

The example uses the helper method **moveElement()**:

```
/**
    moveAFolder

*/
import com.sssw.webdav.client.*;

public class moveAFolder {

    private static boolean m_debug = false;

    public void moveAFolder (String foldersourcepath, String folderdestinationpath)
    {

   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";
   boolean autogen = true; //Generate folders in the folderdestinationpath if they don't
exist

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();

   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);

   //Send the WebDAV request to move folder
   try
   {
      statuscode = dav.moveElement(user, password, realm, foldersourcepath,
folderdestinationpath, autogen);
      if (statuscode==EboDAVStatus.SC_CREATED)
         System.out.println("Request succeeded: The folder " + foldersourcepath + "was
moved to " + folderdestinationpath);
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
```

```
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }

   //End session
   dav.endSession();
   }
}
```

You can also move a resource or collection using the Slide **MoveMethod** class and exteNd Director utility methods. See "Programming practices using utility methods" on page 146.

Removing a category reference from a document

The following code examples show how to remove a category reference from a document. A category is a descriptive name used to group documents logically in the CM subsystem.

Code example: removing a category reference using a helper method

This example uses the helper method **removeCategoryFromDocument()**:

```
/**
    removeCategoryReferenceFromDocument

*/
import com.sssw.webdav.client.*;

public class removeCategoryReferenceFromDocument {

    private static boolean m_debug = false;

    public void removeCategoryReferenceFromDocument (String document, String categoryUUID)
    {

   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();

   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();
```

```
   //Start a session
   dav.startSession("localhost", 80);

   //Send the WebDAV request to remove a category reference
   try
   {
      statuscode = dav.removeCategoryFromDocument(user, password, realm, document,
categoryUUID);
      if (statuscode==EboDAVStatus.SC_MULTI_STATUS)
         System.out.println("Request succeeded: The category was removed from " +
document);
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }

   //End session
   dav.endSession();
   }
}
```

Code example: removing a category reference using utility methods

This example uses the Slide **PropPatchMethod** class and the exteNd Director utility methods **startSession()**, **createCredentials()**, **setCredentials()**, **getState()**, **setState()**, and **createWebDAVMethod()**.

The method that removes the category reference is addPropertyToSet(), called on the PropPatchMethod object. Notice that the second argument—*property value*—is null because the category UUID is passed as the first argument—*property name*. For more information about addPropertyToSet() and its arguments, see "Constructing WebDAV requests that use Proppatch" on page 146.

```
/**
    removeCategoryReference

*/
import com.sssw.webdav.client.*;
import com.sssw.webdav.common.EboWebdavConstants;
import org.apache.webdav.lib.*;
import org.apache.webdav.lib.methods.*;

public class removeCategoryReference
{
   private static boolean m_debug = false;
```

```
   public void removeCategoryReference (String document, String categoryUUID)
   {
   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";
   String namespace-abbr = "RC";

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();

   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);

   //Get and set credentials
   Credentials credentials = dav.createCredentials(user, password);
   dav.setCredentials(credentials);

   //Get and set state and authentication realm
   State state = dav.getState();
   state.setAuthenticateToken(realm);
   dav.setState(state);

   //Create the WebDAV method object PropPatchMethod
   PropPatchMethod ppm =
(PropPatchMethod)dav.createWebdavMethod(dav.PROPPATCH_METHOD,document);
   ppm.addPropertyToSet( categoryUUID, null, namespace-abbr,
EboWebdavConstants.PROPPATCH_REMOVECATEGORY);

   //Execute PropPatchMethod (send the WebDAV request to remove category reference)
   try
   {
      dav.executeCommand(ppm);
      statuscode = ppm.getStatusCode();
      if (statuscode == (EboDAVStatus.MULTI_STATUS))
        System.out.println("Request succeeded: The category was removed from " + document
+ ".");
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }
   catch (java.net.MalformedURLException murle)
   {
```

```
      if (m_debug)
         murle.printStackTrace();
      else
         System.out.println(murle.getMessage());
   }
   catch (java.io.IOException ioe)
   {
      if (m_debug)
         ioe.printStackTrace();
      else
         System.out.println(ioe.getMessage());
   }


   //End session
   dav.endSession();
   }
}
```

Removing all category references from a document

The following code examples show how to remove all category references from a document. A category is a descriptive name used to group documents logically in the CM subsystem.

Code example: removing all category references using a helper method

This example uses the helper method **removeAllCategoriesFromDocument()**:

```
/**
    removeAllCategoryReferencesFromDocument

*/
import com.sssw.webdav.client.*;

public class removeAllCategoryReferencesFromDocument {

    private static boolean m_debug = false;

    public void removeAllCategoryReferencesFromDocument (String document)
    {

   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();
```

```
   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);

   //Send the WebDAV request to remove all category references from the document
   try
   {
      statuscode = dav.removeAllCategoriesFromDocument(user, password, realm, document);
      if (statuscode==EboDAVStatus.MULTI_STATUS)
         System.out.println("Request succeeded: All categories were removed from " +
document);
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }

   //End session
   dav.endSession();
   }
}
```

Code example: removing all category references using utility methods

This example uses the Slide **PropPatchMethod** class and the exteNd Director utility methods **startSession()**, **createCredentials()**, **setCredentials()**, **getState()**, **setState()**, and **createWebDAVMethod()**.

The method that removes all category references is addPropertyToSet(), called on the PropPatchMethod object. Notice that the second argument—*property value*—is null because the document UUID is passed as the first argument—*property name*. For more information about addPropertyToSet() and its arguments, see "Constructing WebDAV requests that use Proppatch" on page 146.

```
/**
    removeAllCategoryReferences

*/
import com.sssw.webdav.client.*;
import com.sssw.webdav.common.EboWebdavConstants;
import org.apache.webdav.lib.*;
import org.apache.webdav.lib.methods.*;

public class removeAllCategoryReferences
```

```
{
   private static boolean m_debug = false;

   public void removeAllCategoryReferences (String documentUUID)
   {
   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";
   String namespace-abbr = "RAC";

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();

   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);

   //Get and set credentials
   Credentials credentials = dav.createCredentials(user, password);
   dav.setCredentials(credentials);

   //Get and set state and authentication realm
   State state = dav.getState();
   state.setAuthenticateToken(realm);
   dav.setState(state);

   //Create the WebDAV method object PropPatchMethod
   PropPatchMethod ppm =
(PropPatchMethod)dav.createWebdavMethod(dav.PROPPATCH_METHOD,document);
   ppm.addPropertyToSet( documentUUID, null, namespace-abbr,
EboWebdavConstants.PROPPATCH_REMOVEALLCATEGORIES);

   //Execute PropPatchMethod (send the WebDAV request to remove all category references)
   try
   {
      dav.executeCommand(ppm);
      statuscode = ppm.getStatusCode();
      if (statuscode == (EboDAVStatus.SC_OK))
         System.out.println("Request succeeded: All categories were removed.");
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }
```

```
   catch (java.net.MalformedURLException murle)
   {
      if (m_debug)
         murle.printStackTrace();
      else
         System.out.println(murle.getMessage());
   }
   catch (java.io.IOException ioe)
   {
      if (m_debug)
         ioe.printStackTrace();
      else
         System.out.println(ioe.getMessage());
   }

//End session
   dav.endSession();
   }
}
```

Renaming a resource or collection

The following code shows how to rename a document. In this case, the source path points to a document. The destination path is identical to the source path, except for a different document name.

To rename other types of resources or collections, make sure the source path points to the element of interest and the destination path points to the same element, but with a different name.

Code example: renaming a document using a helper method

The example uses the helper method **moveElement()**:

```
//**
    renameADocument

*/
import com.sssw.webdav.client.*;

public class renameADocument {

    private static boolean m_debug = false;

    public void renameADocument (String docsourcepath, String docdestinationpath)
    {

   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
```

```
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";
   boolean autogen = false; //Do not generate folders in the docdestinationpath if they
don't exist

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();

   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);

   //Send the WebDAV request to rename the document
   try
   {
      statuscode = dav.moveElement(user, password, realm, docsourcepath,
docdestinationpath, autogen);
      if (statuscode==EboDAVStatus.SC_CREATED)
         System.out.println("Request succeeded: The document " + docsourcepath + "was
renamed to " + docdestinationpath);
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }

   //End session
   dav.endSession();
   }
}
```

You can also rename a resource or collection using the Slide **MoveMethod** class and exteNd Director utility methods. See "Programming practices using utility methods" on page 146.

Setting the value of a custom field in a document

The following code examples show how to update the custom metadata in a document by setting the value of a custom field. Custom fields are fields that you define as part of a custom document type created in the CM subsystem using the CM API or the CMS Administration Console.

To update standard metadata in a document, use the addPropertyToSet() method on a Proppatch method object, as described in "Constructing WebDAV requests that use Proppatch" on page 146.

Code example: setting a field value using a helper method

This example uses the helper method **setFieldValueForDocument()**. This method overwrites existing values of custom fields:

```
/**
    setFieldValueOfADocument

*/
import com.sssw.webdav.client.*;

public class setFieldValueOfADocument {

    private static boolean m_debug = false;

    public void setFieldValueOfADocument (String document, String field_name, String
field_value)
    {

   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();

   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);

   //Send the WebDAV request to update the custom field
   try
   {
      statuscode = dav.setFieldValueForDocument(user, password, realm, document,
field_name, field_value);
      if (statuscode==EboDAVStatus.SC_MULTI_STATUS)
```

```
         System.out.println("Request succeeded: The field " + field_name + " of document "
+ document + "was changed to " + field_value);
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }

   //End session
   dav.endSession();
   }
}
```

Code example: setting a field value using utility methods

This example uses the Slide **PropPatchMethod** class and the exteNd Director utility methods **startSession()**, **createCredentials()**, **setCredentials()**, **getState()**, **setState()**, and **createWebDAVMethod()**:

```
/**
     setTheFieldValue

*/
import com.sssw.webdav.client.*;
import com.sssw.webdav.common.EboWebdavConstants;
import org.apache.webdav.lib.*;
import org.apache.webdav.lib.methods.*;

public class setTheFieldValue
{
   private static boolean m_debug = false;

   public void setTheFieldValue (String document)
   {
   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";
   String fieldname = "Department";
   String fieldvalue = "Human Resources";
   String namespace-abbr = "SFV";

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();

   //Instantiate an EboDAVStatus object
```

```
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);

   //Get and set credentials
   Credentials credentials = dav.createCredentials(user, password);
   dav.setCredentials(credentials);

   //Get and set state and authentication realm
   State state = dav.getState();
   state.setAuthenticateToken(realm);
   dav.setState(state);

   //Create the WebDAV method object PropPatchMethod
   PropPatchMethod ppm =
(PropPatchMethod)dav.createWebdavMethod(dav.PROPPATCH_METHOD,document);
   ppm.addPropertyToSet( fieldname, fieldvalue, namespace-abbr,
EboWebdavConstants.PROPPATCH_SETFIELDVALUE);

   //Execute PropPatchMethod (send the WebDAV request to set field value)
   try
   {
      dav.executeCommand(ppm);
      statuscode = ppm.getStatusCode();
      if (statuscode == (EboDAVStatus.SC_MULTI_STATUS))
         System.out.println("Request succeeded: The field " + fieldname + " was set to " +
fieldvalue + ".");
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }
   catch (java.net.MalformedURLException murle)
   {
      if (m_debug)
         murle.printStackTrace();
      else
         System.out.println(murle.getMessage());
   }
   catch (java.io.IOException ioe)
   {
      if (m_debug)
         ioe.printStackTrace();
      else
         System.out.println(ioe.getMessage());
   }
```

```
   //End session
   dav.endSession();
   }
}
```

Unlocking a document

The following code shows how to unlock a document, making it available to other authors in a collaborative environment. You might invoke this function in your WebDAV client application when a user checks in a document.

Code example: unlocking a document using a helper method

The example uses the helper method **unlockDocument()**. This method throws an exception if the document of interest is already unlocked. To explicitly check for this condition, the code calls the checkLockToken() method:

```
/**
    unlockADocument

*/
import com.sssw.webdav.client.*;

public class unlockADocument {

    private static boolean m_debug = false;

    public void unlockADocument (String document)
    {

    //Define variables
    int statuscode = 0;
    String user = "contentadmin";
    String password = "contentadmin";
    String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";

    //Instantiate an EboDAVSwitch object
    EboDAVSwitch dav = new EboDAVSwitch();

    //Instantiate an EboDAVStatus object
    EboDAVStatus status = new EboDAVStatus();

    //Start a session
    dav.startSession("localhost", 80);

       try
      {
        //If document is locked, unlock it
        if (dav.checkLockToken(document) != null)
        {
```

```
            //Send the WebDAV request to unlock the document
            statuscode = dav.unlockDocument(user, password, realm, document);
            if (statuscode==EboDAVStatus.SC_NO_CONTENT)
               System.out.println("Request succeeded: The document " + document + "was
unlocked.");
            else
               System.out.println("Request failed: " + status.getStatusText(statuscode));
         }
          else
         {
            System.out.println("The document is already unlocked.");
         }
      }
       catch (EboDAVException e)
      {
         if (m_debug)
            e.printStackTrace();
         else
            System.out.println(e.getMessage());
         }
      }
      //End session
      dav.endSession();
   }
}
```

You can also unlock a document using the Slide **UnlockMethod** class and exteNd Director utility methods. See "Programming practices using utility methods" on page 146.

Updating a document

The following code example shows how to update the content of a document.

Code example: updating a document using a helper method

The example uses the helper method **putDocument()**. This method updates the content—not the metadata—of a document by creating and publishing a new version. To update document metadata, see "Setting the value of a custom field in a document" on page 177.

```
/**
    updateADocument

*/
import com.sssw.webdav.client.*;

public class updateADocument {

    private static boolean m_debug = false;
```

```
    public void updateADocument (String document)
    {

   //Define variables
   int statuscode = 0;
   String user = "contentadmin";
   String password = "contentadmin";
   String realm = "Basic realm = \"SSSW_WEBDAV_AUTHENTICATION\"";
   String updatetext = "Hello earth!";
   byte [] newcontent = updatetext.getBytes();

   //Instantiate an EboDAVSwitch object
   EboDAVSwitch dav = new EboDAVSwitch();

   //Instantiate an EboDAVStatus object
   EboDAVStatus status = new EboDAVStatus();

   //Start a session
   dav.startSession("localhost", 80);

   //Send the WebDAV request to update the document
   try
   {
      statuscode = dav.putDocument(user, password, realm, document, newcontent);
      if (statuscode==EboDAVStatus.SC_OK)
         System.out.println("Request succeeded: The document " + document + "was
updated.");
      else
         System.out.println("Request failed: " + status.getStatusText(statuscode));

   }
   catch (EboDAVException e)
   {
      if (m_debug)
         e.printStackTrace();
      else
         System.out.println(e.getMessage());
   }

   //End session
   dav.endSession();
   }
}
```

You can also update a document or create a new document using the Slide **PutMethod** class and exteNd Director utility methods. See "Programming practices using utility methods" on page 146.

11 Working with WebDAV Events

This chapter describes how to handle events related to WebDAV operations and activities. It has these sections:

- About WebDAV events
- Registering for WebDAV events
- Enabling WebDAV events

 This chapter assumes familiarity with the exteNd Director event model and event handling. For more information, see the chapter on working with events in *Developing exteNd Director Applications*.

About WebDAV events

WebDAV subsystem events are an extension of the base exteNd Director event model framework, consisting of *state change events*, *event producers*, and *event listeners* (including vetoable listeners). The API for WebDAV events is defined in the com.sssw.webdav.event.api package.

Event types

The API defines a set of state changed events related to WebDAV operations. Event IDs are exposed on the individual event classes as well as on the com.sssw.webdav.event.api.EbiConstants interface. There are also generic state change events defined in com.sssw.fw.event.api.EbiotateChangeEvent.

Below is a list of event IDs defined in com.sssw.webdav.event.api.EbiConstants:

📖 For more information about WebDAV operations, see "Supported WebDAV methods" on page 134.

WebDAV operation	Event ID constant
Copy collections and resources	COPY_EVENT_ID
Delete collections or resources	DELETE_EVENT_ID
Retrieve collections or resources	GET_EVENT_ID
Retrieve header only	HEAD_EVENT_ID
Create a lock specified by the lockinfo XML element on the Request-URI.	LOCK_EVENT_ID
Create collection	MKCOL_EVENT_ID
Move resources or collections	MOVE_EVENT_ID
Return methods that can be called on resources and collections	OPTIONS_EVENT_ID
Download resources and collections from the client	POST_EVENT_ID
Retrieve properties on resources and collections	PROPFIND_EVENT_ID
Set and/or remove properties on server-side resources and collections	PROPPATCH_EVENT_ID
Upload resources and collections from the client	PUT_EVENT_ID
Remove a lock identified in the Lock-Token request header of the Request-URI.	UNLOCK_EVENT_ID

Registering for WebDAV events

➢ **To subscribe to WebDAV events:**

- Use the addStateChangeListener() or add VetoableStateChangeListener method available on the EbiStateChangeProducer interface.

You can register for a specified type or types of events using this version of addStateChangeListener():

```
public boolean addStateChangeListener(
   BitSet events, EbiStateChangeListener listener)
```

where *events* is a bit set of event IDs.

Use the event IDs specified in com.sssw.webdav.event.api.EbiConstants. For example, this code registers for create, delete and move operations on collections and resources:

```
EbiStateChangeProducer producer = new EbiStateChangeProducer()
// Instantiate a Java BitSet and populate it
BitSet events = new BitSet();
events.set(EbiConstants.MKCOL_EVENT_ID_ID);
events.set(EbiConstants.DELETE_EVENT_ID);
events.set(EbiConstants.MOVE_EVENT_ID);
// Add listener
producer.addStateChangeListener(events, Mylistener);
```

Enabling WebDAV events

➢ **To enable or disable WebDAV events:**

1. Open config.xml for WebDAV in your exteNd Director project.
2. Find this property:

   ```
   com.sssw.webdav.events.enable.Default
   ```

3. Set the value to **true** for enable or **false** for disable.
4. Redeploy your project.

III CMS Administration Console

Describes how to use the CMS Administration Console, a graphical user interface for developing and managing a content management scheme

- Chapter 12, “About the CMS Administration Console”
- Chapter 13, “Setting Up the Required Infrastructure”
- Chapter 14, “Setting Up the Optional Infrastructure”
- Chapter 15, “Creating Content”
- Chapter 16, “Maintaining Content”
- Chapter 17, “Administering Content”
- Chapter 18, “Searching Content”
- Chapter 19, “Managing Content Security”
- Chapter 20, “Importing and Exporting Content”
- Chapter 21, “Administering Automated Tasks”

12 About the CMS Administration Console

This chapter describes what tasks you can perform with the Content Management Subsystem Administration Console, or CMS Administration Console. It has these sections:

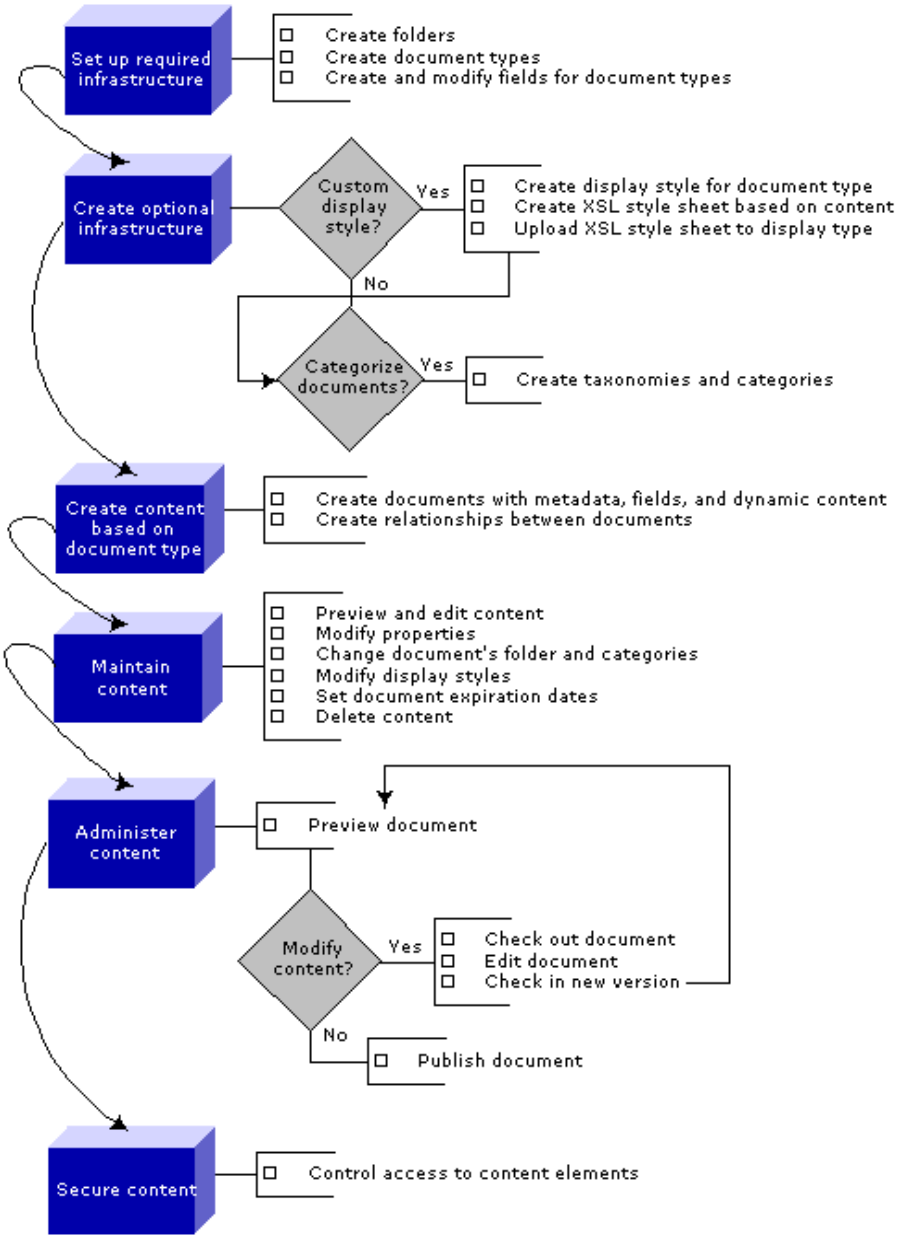
- What CM tasks you can do with the CMS Administration Console
- How to access the CMS Administration Console
- The main CMS Administration Console page

IMPORTANT: Along with exteNd Director, you must have Microsoft Internet Explorer Version 5.5 or higher installed for running the CMS Administration Console.

What CM tasks you can do with the CMS Administration Console

You can use the CMS Administration Console to perform all tasks related to managing content throughout its dynamic life cycle in the exteNd Director application.

The following diagram presents the recommended order and interaction of these tasks during a typical CMS Administration Console session:



📖 For more information on how to perform these tasks with the CMS Administration Console, see these sections:

- Chapter 13, “Setting Up the Required Infrastructure”
- Chapter 14, “Setting Up the Optional Infrastructure”
- Chapter 15, “Creating Content”
- Chapter 16, “Maintaining Content”
- Chapter 17, “Administering Content”
- Chapter 19, “Managing Content Security”

Importing and exporting documents In addition to the CM tasks shown in the diagram above, the CMS Administration Console allows you to import and export documents.

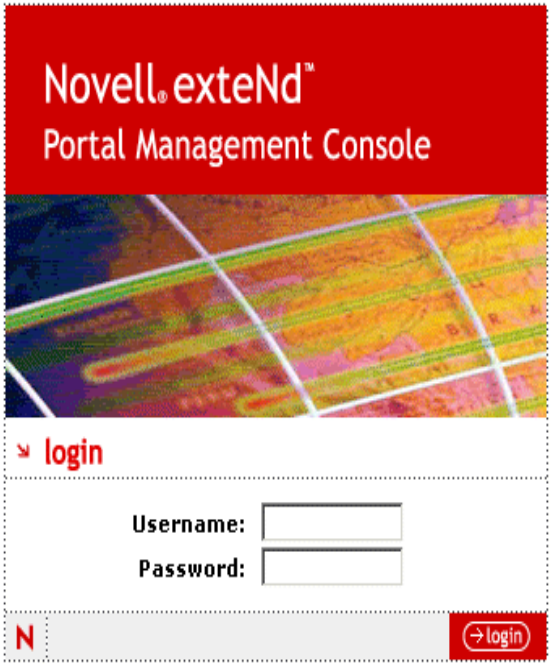
📖 For more information on how to import and export documents using the CMS Administration Console, see Chapter 20, “Importing and Exporting Content”.

How to access the CMS Administration Console

You can access the CMS Administration Console by selecting **Content Management** from the Director Administration Console (DAC).

📖 For information about how to access the DAC, see the section on accessing the DAC in *Developing exteNd Director Applications*.

From an EAR project, the CMS Administration Console login page looks like this:



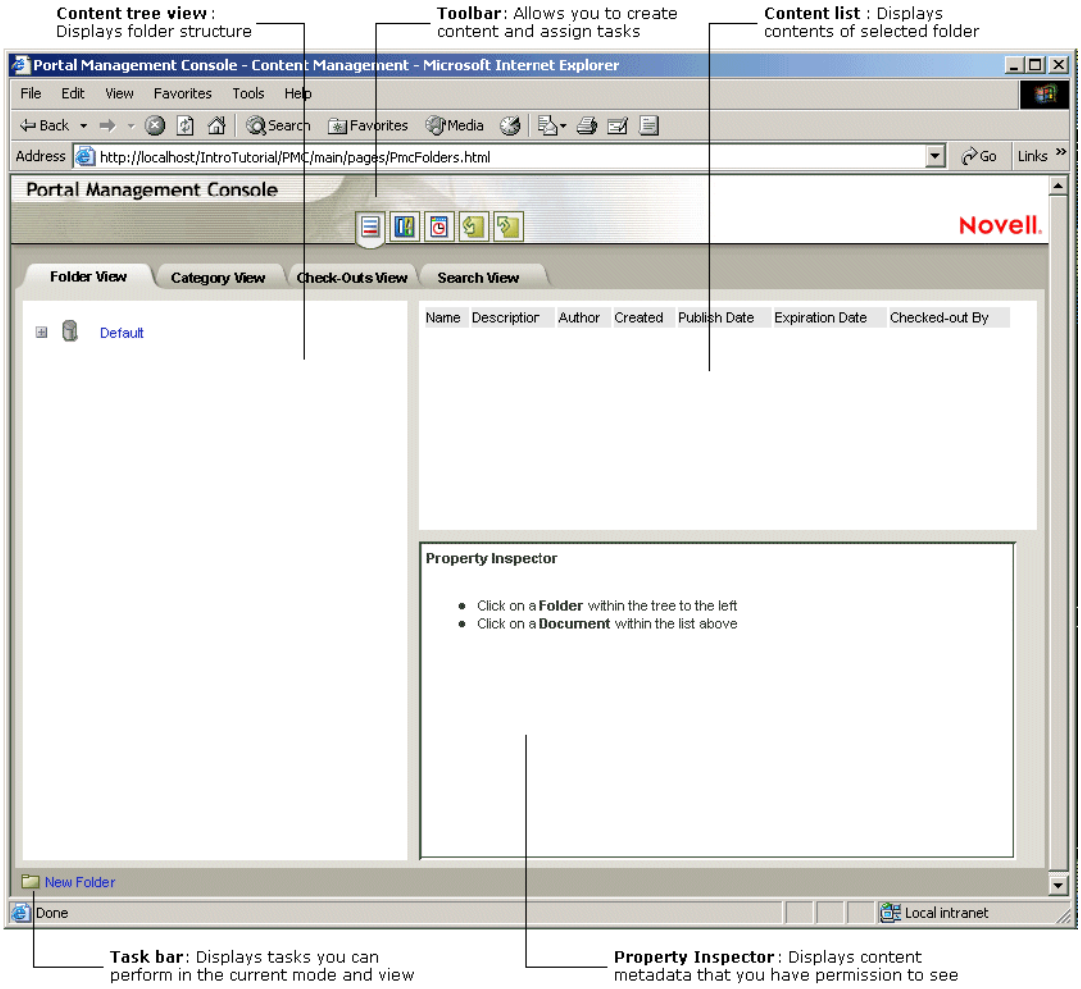
Log in by entering your user name and password and then clicking **OK**.

NOTE: Check with your administrator to make sure you have the necessary user privileges for performing the CM tasks assigned to you. For more information, see Chapter 19, "Managing Content Security".

The main CMS Administration Console page opens in your browser, as described in the next section.

The main CMS Administration Console page

When you start the CMS Administration Console, the main page appears—as in this example:



The CMS Administration Console has several views and modes that you control via interactive controls—as follows.

Interactive controls

The CMS Administration Console consists of the following interactive controls:

- Toolbar
- Content view tabs
- Content tree view
- Content list
- Context-sensitive toolbar
- Property Inspector

Toolbar—switch between modes

To switch between **modes**:

Mode	Icon	What authorized users can do
Content		Set up content infrastructure, and administer and secure content, search for documents By default, the CMS Administration Console opens in content mode, displaying your content by container in the **content tree view** and by document in the **document list**.
Templates		Define document types, display styles, and fields—and create content based on these specifications
Tasks		View, start, and stop automated tasks
Import		Import content infrastructure, documents, document types, display styles, and fields
Export		Export content infrastructure, documents, document types, display styles, and fields

Content view tabs—display views of content infrastructure

To display **views** of the content infrastructure:

View	Displays
Folder	Physical content infrastructure as a tree view of folders
Category	Logical content infrastructure as a tree view of taxonomies and categories
Check-Outs	Documents checked out, by either the current user or other users

View	Displays
Search	Search dialogs and documents found by the most recent search

For more information about folders and categories, see "Subsystem infrastructure" on page 21. For information about checking out documents, see "Checking documents in and out" on page 260. For information about finding documents with the Search facility, see Chapter 18, "Searching Content".

Content tree view

Displays:

- The **physical** infrastructure in **folder** view
- The **logical** infrastructure in **category** view.

Content list

Displays the list of documents in the selected folder, along with identifying information such as name, author, description, create date, expiration date, publish date, and checkout status.

Context-sensitive toolbar

Provides functions based on the current mode and view.

Property Inspector

Displays properties for selected documents, folders, taxonomies, and categories.

The Property Inspector is context-sensitive and permission-sensitive. It displays interactive controls and tabbed panes of information based on the object you select and the permissions associated with your user ID.

For example: if you do not have WRITE permission, you cannot edit documents and the Property Inspector will not display Check-In and Check-Out controls; if you do not have PROTECT permission, you cannot set security on content and the Property Inspector will not display a Security tab.

13 Setting Up the Required Infrastructure

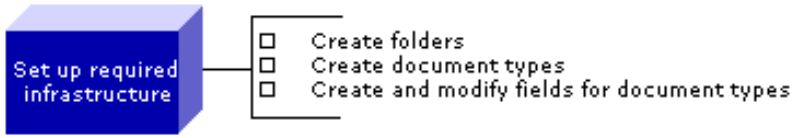
This chapter describes the order of tasks required for setting up the required parts of the infrastructure, along with associated procedures. It has these sections:

- Flow of operations
- Creating folders
- Creating document types
- Creating fields and adding them to a document type
- Writing JavaScript for document types and fields

NOTE: Before creating documents for your exteNd Director application, you must define the content infrastructure, as described in "Subsystem infrastructure" on page 21.

Flow of operations

Here is a workflow that illustrates the recommended order of operations for setting up the required parts of the Content Management (CM) subsystem infrastructure:



Generally, the task of building this infrastructure is assigned to a system administrator or content administrator who has READ, WRITE, and LIST permissions. For more information about managing security, see Chapter 19, "Managing Content Security".

Creating folders

The folder is a key part of the CM subsystem. Every document must reside in one (and only one) folder, although a single folder can store one or more documents as well as other folders.

➢ **To create a folder:**

1. Enter Content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your existing folders appear in the content tree view.
3. Select the folder that will house your folder by clicking the name.

   The name appears highlighted.
4. Click the **New Folder** icon, located in the bottom-left panel of the CMS Administration Console.

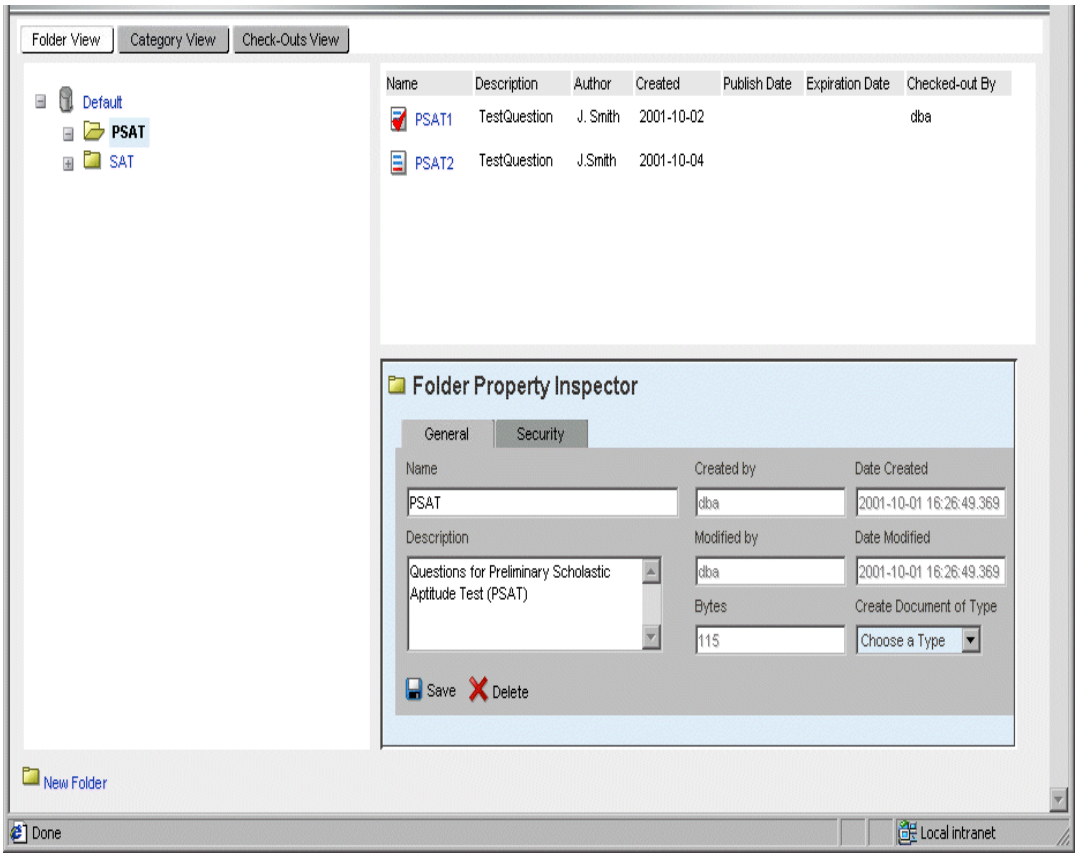
   An Untitled folder appears in the content tree view.

   You may have to expand the parent folder in the content tree view to make the new folder visible in that view.
5. Click **Untitled** to open the Property Inspector for the new folder.
6. Fill in the **Name** and **Description** text boxes in the Property Inspector, then click **Save**.

   The other General fields are filled in automatically by the CMS Administration Console. You cannot edit them.
7. Select the **Security** tab in the Property Inspector and set security for the folder, as described in Chapter 19, “Managing Content Security”.
8. Click **Save** to preserve your settings.
9. Select the folder in the content tree view.

   Your new folder should appear in the content tree view as well as in the content list along with the description, author, and date created.

Here is an example showing information about a PSAT folder:



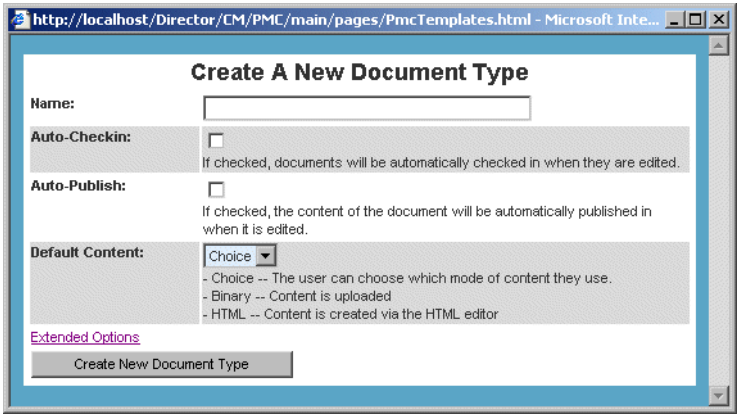
Creating document types

A *document type* is the basic definition of a document. Every document is associated with a document type in the CMS Administration Console.

The document type is a template that specifies layout styles, fields of information, and document management options—such as whether or not the CMS Administration Console automatically checks in a document after it is edited.

➢ **To create a document type:**

1. Enter Templates mode by clicking the **Templates** button in the toolbar.
   A panel appears listing any document types that have been defined.
2. Click the **Add** button that appears under the Document Types list.
   The Create A New Document Type window appears:

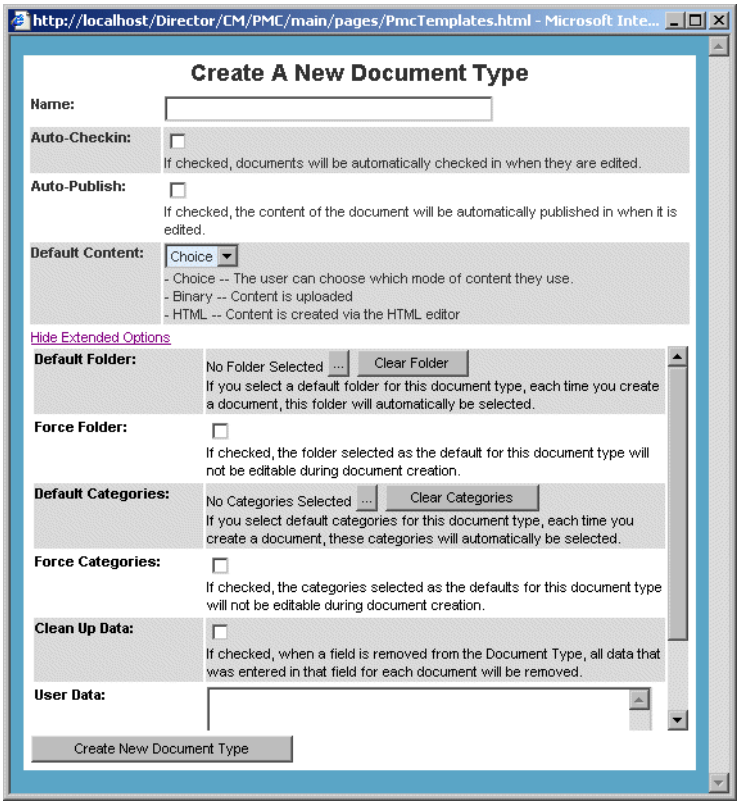


3. Specify the basic options, including:

Option	Effect
Auto-Checkin	If selected, CMS Administration Console checks in documents automatically after they are edited. If not selected, CMS Administration Console does **not** check in documents automatically after they are edited
Auto-Publish	If selected, CMS Administration Console publishes the latest version of the content of a document automatically after that document is edited. If not selected, CMS Administration Console does **not** publish documents automatically after they are edited
Default Content	If you select: **HTML**: CMS Administration Console will always enter content as HTML for documents of this type. **Binary**: CMS Administration Console will always upload content from an external source for documents of this type. **Choice**: You want to decide at content creation time whether to enter content as HTML or upload content from an external source.

4 Click **Extended Options** to specify additional document type behavior.

The Create A New Document Type window expands:



5 Specify extended options, including:

Option	Effect
Default Folder	When the CMS Administration Console creates documents of this type, this folder is specified as the parent folder. You can change the folder when creating the new document.
Force Folder	If selected, the folder specified under Default Folder cannot be changed when creating a new document of this type.
Default Categories	When the CMS Administration Console creates documents of this type, this category is specified as the parent category. You can change the category when creating the new document.

Option	Effect
Force Categories	If selected, the category specified under Default Category cannot be changed when creating a new document of this type.
Clean Up Data	If selected, when you remove a field from a document type (but leave it available for later use), the CMS Administration Console deletes the field from legacy documents of that type. If not selected, when you remove a field from a document type (but leave it available for later use), the CMS Administration Console preserves the field in legacy documents of that type but does not allow you to edit the field.
User Data	You can use the text box to store additional metadata about the document type (such as notes, procedural instructions, and so on).

6 Click the **Create New Document Type** button.

Your new document type is added to the list.

Creating fields and adding them to a document type

About fields

Fields are application-specific metadata that you define as part of a *document type*.

You can create custom fields using the CMS Administration Console or programmatically using the CM API.

NOTE: You must be a member of the SearchAdmin group to create fields. For more information about users and groups, see the chapter on using the Directory section of the DAC in the *User Management Guide*.

You assign each field a control type. The control type you select should reflect the way you'd like the content developer to enter information in the document type template. Each control type requires its own set of parameters, which you can specify in the Property Inspector. When fields are created, they are added to a pool of available fields that are shared by multiple document types.

When you add a field to a document type, an equivalent blank field is added to documents of that type that you have already created in the CMS Administration Console.

Creating and manipulating fields

This section explains how to create fields, add existing fields to document types, and specify which fields to display in the Available Fields list.

➢ **To create a field:**

1. Make sure you are a member of the SearchAdmin group.

   For information, see the procedures described in the chapter on using the Directory section of the DAC in the *User Management Guide*.

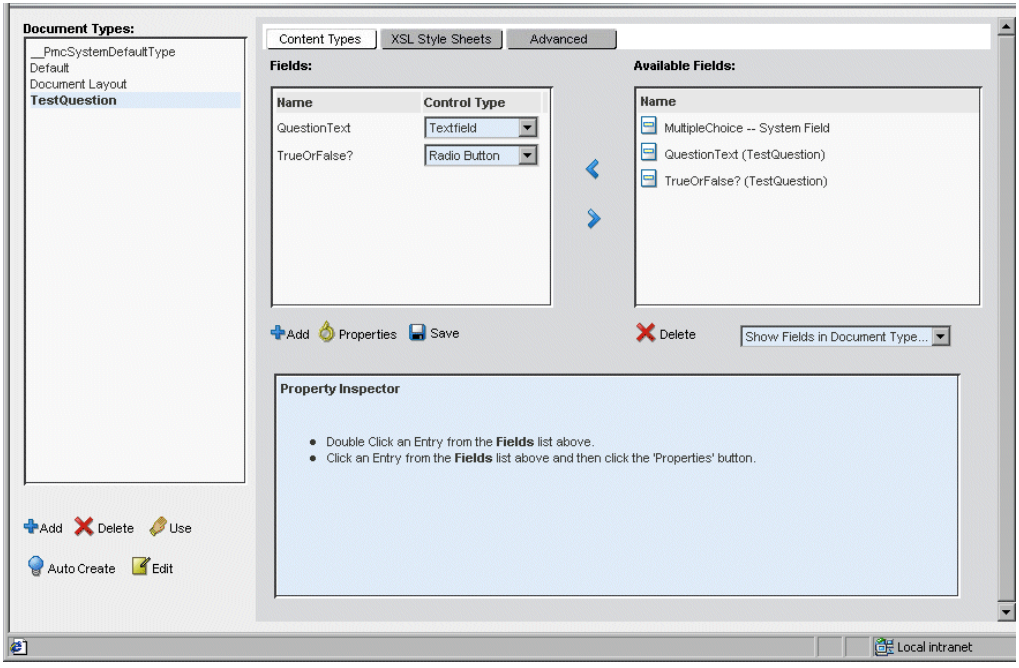
2. Enter templates mode by clicking the **Templates** button in the toolbar.

   A panel appears listing the document types that have been defined.

3. Click the document type for which you are going to create a field.

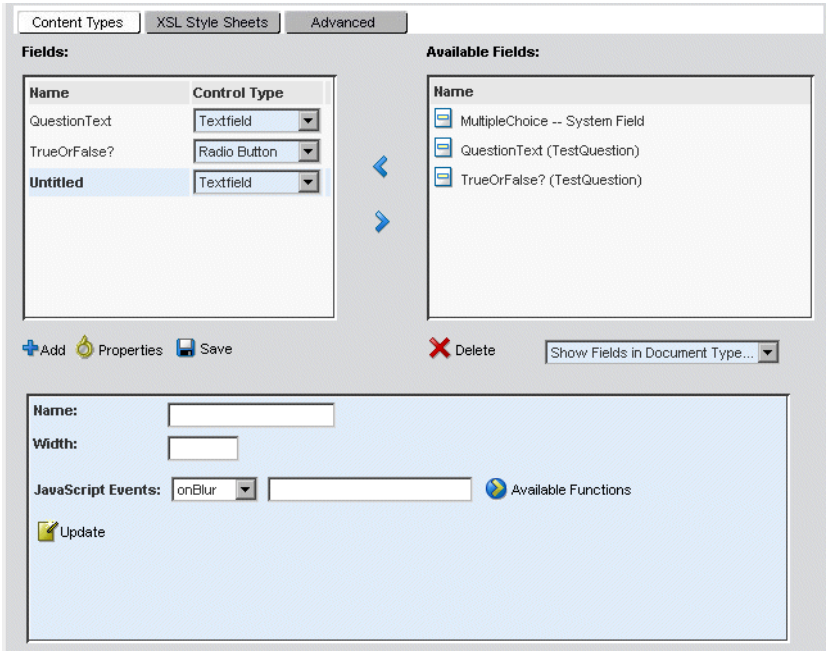
   NOTE: If you want to create a new document type first, see "Creating document types" on page 199.

   A Content Types panel appears displaying the currently defined fields in the document type and providing controls for creating new fields or adding existing fields:



4 Click **Add** in the Content Types panel.

An Untitled field appears in the Fields pane for the selected document type, and the Property Inspector opens allowing you to specify properties for the new field:



5 In the **Fields** pane, select the control type you want for your field. Choices include **Textfield**, **Checkbox**, **Radio Button**, and so on.

The Property Inspector refreshes to display options appropriate for the control type you select. These control types represent HTML control types, and the display options represent the attributes for those control types.

6 In the Property Inspector, enter an informative name for your field and fill in the other parameters.

7 Click **Update**.

The new field appears in the Fields pane for the selected document type and in the Available Fields pane for other document types to use.

8 Repeat these steps for as many fields as you want to create and add to the document type.

9 Click **Save** in the Fields pane to save the fields in the current document type.

➢ **To add an existing field to a document type:**

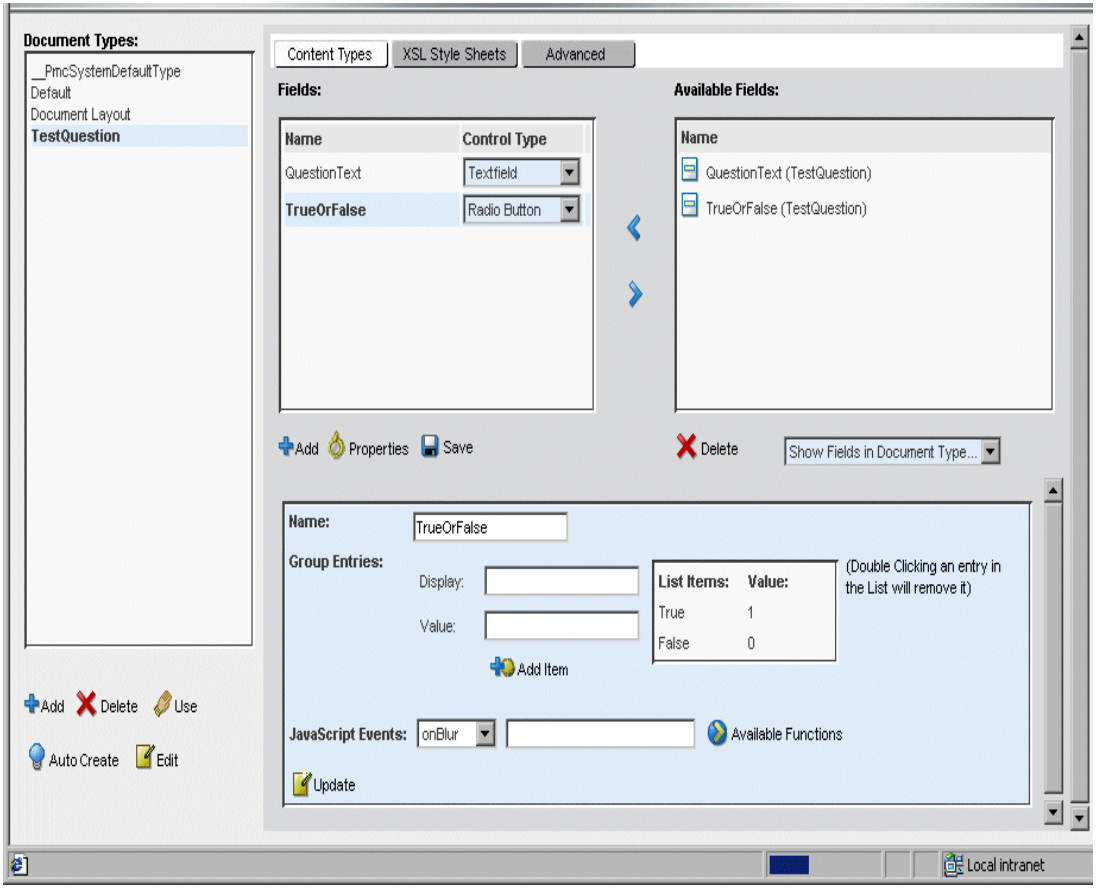
1 Enter templates mode by clicking the **Templates** button in the toolbar.

A panel appears listing all document types that are currently defined.

2 Click the document type for which you want to add a field.

NOTE: If you want to create a new document type first, see "Creating document types" on page 199.

The Content Types panel appears displaying a pane of available fields:



3 Add fields to the document type using one of these methods:

- Double-click the field name in the Available Fields list.

  OR

- Select a field in the Available Fields list and click the **Add Field** button:

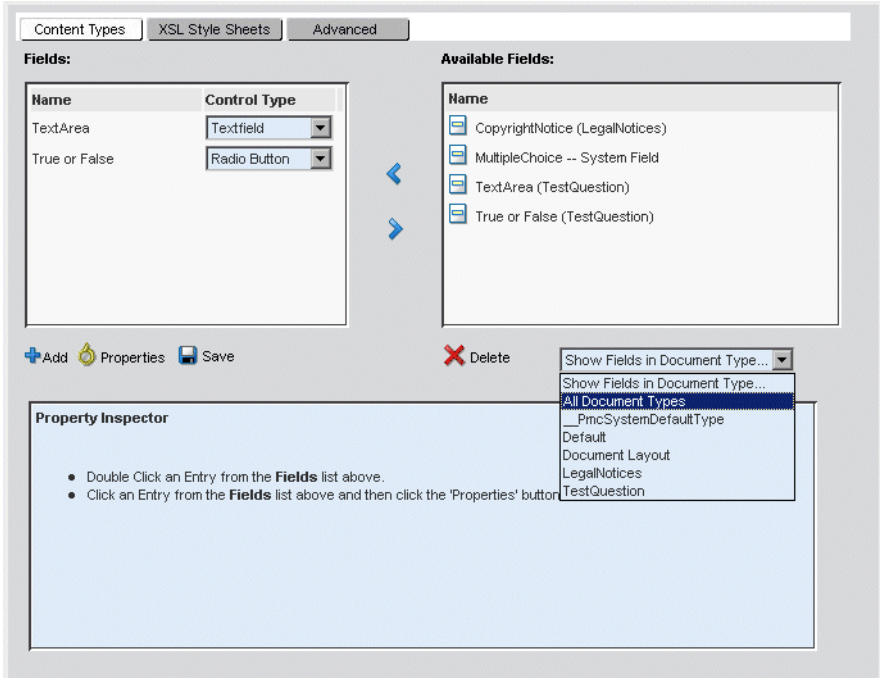
  OR

- Drag the field icon from the Available Fields list to the Fields list:

➢ **To change the Available Fields display:**

1. Click the down arrow of the dropdown menu labeled **Show Fields in Document Type**, located under the Available Fields list. A menu appears allowing you to display the fields available for only a particular document type or for all document types:



2. Select a menu option.

   The Available Fields list refreshes to reflect your choice.

Writing JavaScript for document types and fields

The CMS Administration Console enables you to specify JavaScript code for document types and fields. You can specify JavaScript that runs when:

- A content page is loaded
- An HTML form on a page is submitted
- A field on a page gains or loses focus, or is clicked
- The content of a field is changed

If you code JavaScript for a particular document type, you can access that code when defining JavaScript for individual fields in that document type. For example, if you define a function for the document type, you can call that function on a JavaScript event for a field, such as gaining focus or clicking.

CAUTION: *The CMS Administration Console does not verify JavaScript code. You are responsible for verifying that JavaScript written for a document type or field is designed and coded correctly.*

➢ **To specify JavaScript for a document type:**

1. Enter templates mode by clicking the **Templates** button in the toolbar.

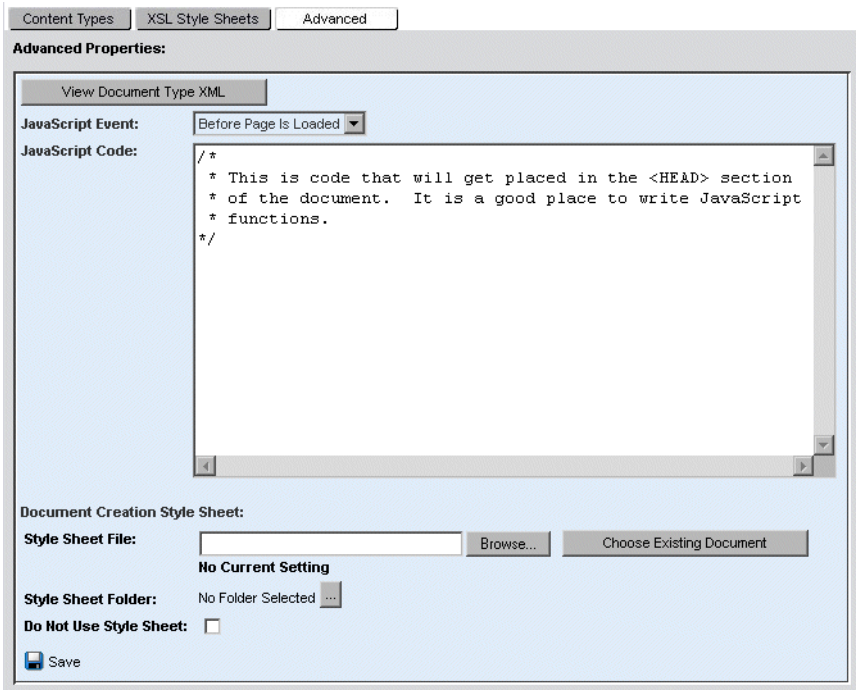
   A panel appears listing all document types currently defined.

2. Click the document type for which you want to specify JavaScript code.

   NOTE: If you want to create a new document type first, see "Creating document types" on page 199.

3. Click the **Advanced** tab.

   The Advanced Properties window displays:



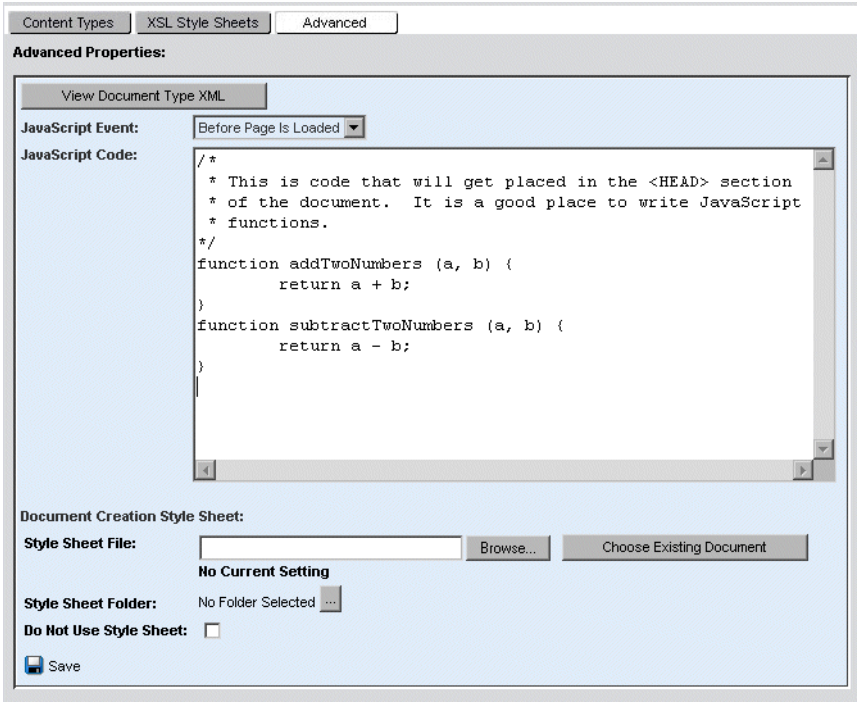
4 Under **JavaScript Event**, specify when you want the JavaScript to run during the life cycle of the document. Choices include:

- Before Page Is Loaded
- After Page Is Loaded
- Form Submitted

If you want the JavaScript code to be available to fields in the document type (for example, if you want to define functions that will be called by individual fields), specify **Before Page Is Loaded**.

5 Under **JavaScript Code**, insert the code.

For example, here is some JavaScript code containing two function definitions that is to run before the page is loaded:



6 Click **Save** to save the JavaScript specification in the current document type.

To code additional JavaScript for other events, repeat this procedure specifying the alternate event(s) in Step 4 and code in Step 5.

➢ **To specify JavaScript for a field:**

1 Enter templates mode by clicking the **Templates** button in the toolbar.

A panel appears listing all document types that are currently defined.

2 Click the document type that contains the field for which you want to specify JavaScript code. A list of the fields defined for that type appears.

NOTE: If you want to create a new document type first, see "Creating document types" on page 199.

3 Double-click the field for which you want to specify JavaScript to access the field properties.

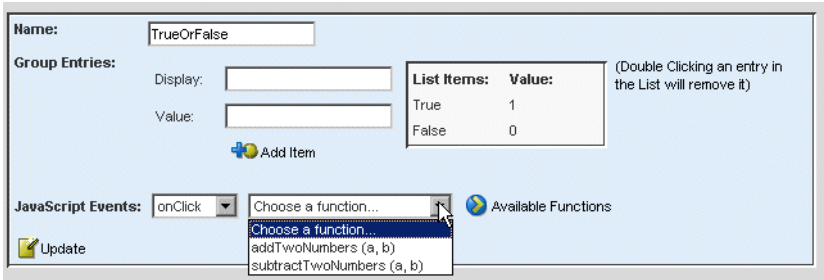
4 Under **JavaScript Events**, specify when you want the JavaScript to run. Depending on the kind of field (text field, check box, text area, and so on) selected, one or more of these events might be available:

- onBlur
- onFocus
- onClick
- onChange

You can specify different JavaScript code for different events.

5 In the text box next to the **JavaScript Events** selection box, type your JavaScript code.

If any functions for the document type that contains the field have been defined, you can click **Available Functions** to select from the list of predefined functions:



A template for the function is inserted into the text box. You can then edit the text box.

For information about defining JavaScript functions for a document type, see "To specify JavaScript for a document type:"above.

6 Click **Update** to save your field properties.

To code additional JavaScript for other field events, repeat this procedure specifying the alternate event(s) in Step 4 and code in Step 5.

CAUTION: *If you create a field that references a function defined in a particular document type and then use that field in another document type, you must redefine the function in the second document type before that function can work.*

---

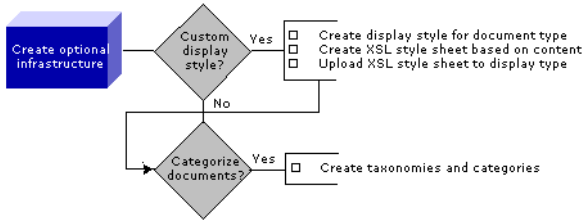
14 Setting Up the Optional Infrastructure

In addition to the required infrastructure such as document types and folders, you can create optional parts of the CM subsystem infrastructure that define display styles and assign categories to content. This chapter has these sections:

- Flow of operations
- Creating display styles
- Specifying a style sheet for a document type
- Creating taxonomies
- Creating categories

Flow of operations

Here is a workflow that illustrates the recommended order of operations for setting up the optional CM subsystem infrastructure:



Creating display styles

About display styles

Display styles specify how to display content for individual document types. The CMS Administration comes with a default display style that it automatically applies to all content unless you override it by creating custom display styles for document types.

For each display style, you can add one or more XSL style sheets that specify how to render content for particular user agents, such as Microsoft Internet Explorer and Netscape Navigator. You must create the XSL specifications in an external XSL editor, then upload the XSL file to a display style.

The CMS Administration Console treats XSL style sheets like documents—by:

- Storing each XSL style sheet in one (and only one) folder, identifying it as a system *resource*
- Storing each update to an XSL style sheet as a new version
- Requiring authorized users to publish the version of the XSL style sheet they want to apply to content

Before you create display styles Before you can create display styles, the following elements of the content infrastructure must be in place:

Infrastructure element	For information see
Folder for physically storing XSL style sheets	"Creating folders" on page 198
Document type for defining content structure	"Creating document types" on page 199 and "Creating fields and adding them to a document type" on page 202
Instances of the document type(s) for which you want to create a display style	"Creating documents" on page 223

Creation procedure After you complete these tasks, you are ready to:

- Create an XSL style sheet in an external editor based on existing content
- Create a display style
- Upload the XSL style sheet to the display style

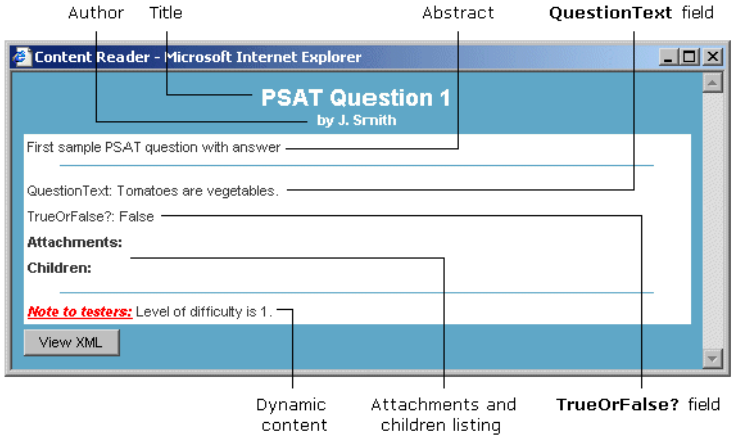
➢ **To create an XSL style sheet based on existing content:**

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your existing folders appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Click to select the folder that contains the content of interest.

   A list of documents appears in the content list.
4. Select the document of interest to open its Property Inspector.
5. Click **Preview** in the Property Inspector.

   The content opens in a Content Reader window:



6. Click **View XML**.

   The Content Reader refreshes to display the XML code that underlies your content, along with a Show Styled Document button that allows you to redisplay the rendered content.
7. Copy the XML and paste it into an XSL editor and develop an XSL style sheet for the content.
8. Save the XSL style sheet in an XSL file on your local file system or designated network directory.

Now you are ready to create a display style that will use the XSL style sheet you just created.

➢ **To create a display style:**

1. Enter templates mode by clicking the **Templates** button in the toolbar.

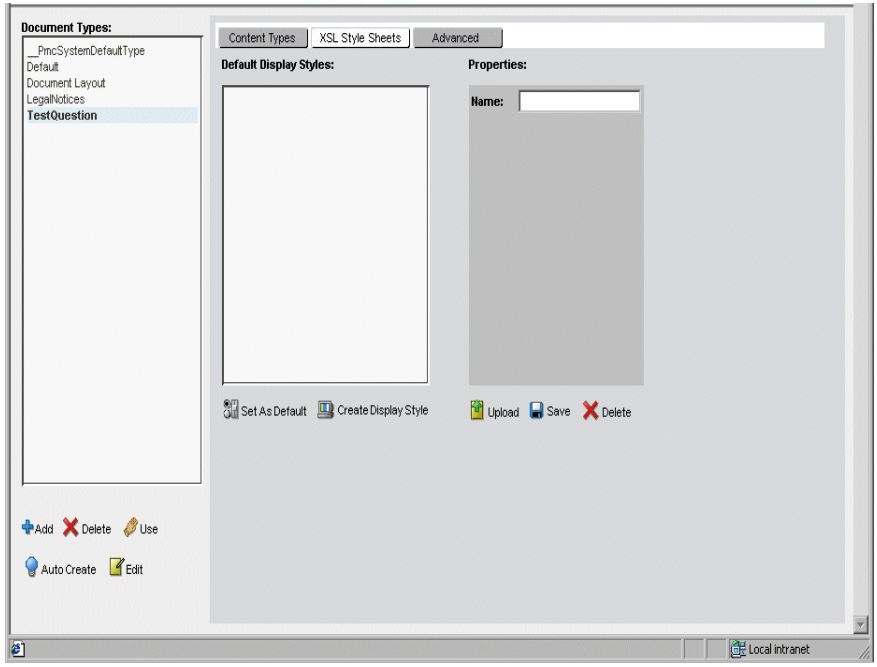
   A panel appears listing the document types that have been defined.

2. Select the document type for which you are going to define a display style.

   TIP: If you want to create a new document type first, see "Creating document types" on page 199.

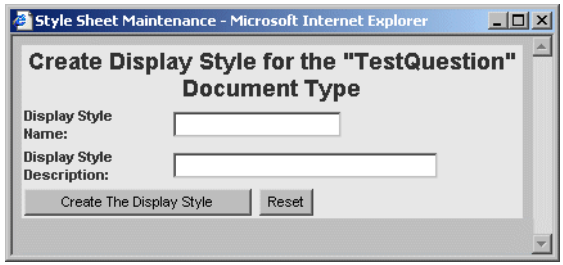
3. Click the **XSL Style Sheets** tab.

   Two panes appear. The Default Display Styles pane lists any display styles that have already been created for the document type, and the Properties pane displays the properties of a selected display style. In the following example, no display styles have been created:



4 Click **Create Display Style**.

The Create Display Style window opens:



5 Enter a name for the new display style and (optionally) a description, then click **Create The Display Style**.

The new display style is added to the Default Display Styles pane.

6 If you want to designate the display style as the default for the selected document type, select the display style in the Default Display Styles pane and click **Set As Default**.

➢ **To upload an XSL style sheet to a display style:**

Before performing this procedure, you must create an XSL style sheet in an external editor and store the specification as an XSL file on your network.

1 Enter Templates mode by clicking the **Templates** button in the toolbar.

A panel appears listing all document types that have been defined.

2 Select the document type that contains the display style of interest.

The document type Property Inspector appears.

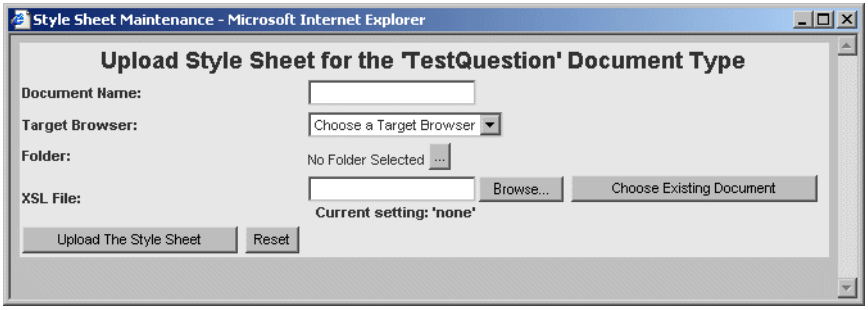
3 Click the **XSL Style Sheets** tab.

Two panes appear. The Default Display Styles pane lists any display styles that have been created for the document type, and the Properties pane displays the properties of a selected display style.

4 In the Default Display Styles pane, select the display style for which you want to add an XSL style sheet.

5 Click **Upload** in the Properties pane to upload the XSL style sheet you created externally.

The Upload Style Sheet window opens:



6 Fill in the text boxes as follows:

Option	What to enter
Document Name	Name that identifies the XSL style sheet in the CMS Administration Console **NOTE:** The CMS Administration Console uses this name to display the XSL style sheet as a document in folder view
Target Browser	A user agent from the dropdown list **NOTE:** The CMS Administration Console uses this value to determine which XSL style sheet should render content for specific user agents
Folder	Folder where the XSL style sheet should be stored
XSL File	XSL style sheet you created for this display style. You can: ◆ Browse the network for an external file OR ◆ Select **Choose Existing Document** to search for an XSL file that has already been uploaded to the CMS Administration Console

7 Click **Upload The Style Sheet**.

The XSL style sheet is uploaded to the display style. If you expand the display style in the Default Display Styles pane, you will see its list of associated XSL style sheets.

The XSL style sheet is also uploaded as a system resource to the folder you specified in Step 6.

Specifying a style sheet for a document type

The properties of a document type can include an XSL style sheet that you can specify on the **Advanced** tab of the document type properties. This style sheet designation is included in the XML of all documents of this type that you create.

The CMS Administration Console content creation code uses this style sheet to render the data for that document type.

This style sheet designation is distinct from the styles and style sheets you can specify in the XSL Style Sheets tab (as described under "Creating display styles" on page 212). Those styles are used when displaying portlets of the document type in the Content Reader.

➢ **To specify a style sheet for the document type:**

1. Enter templates mode by clicking the **Templates** button in the toolbar.

   A panel appears listing all document types that have been defined.

2. Select the document type that contains the display style of interest.

   A document type Property Inspector appears.

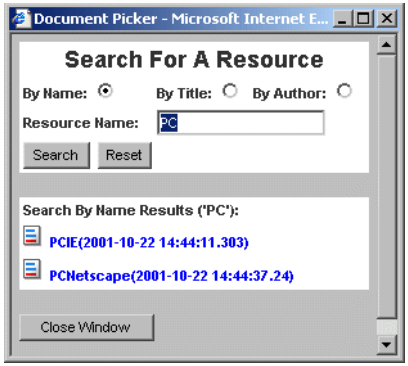
3. Click the **Advanced** tab.

4. To specify a style sheet document that currently exists in the CMS Administration Console:

   4a Click **Choose Existing Document**.

   The Search For A Resource window opens.

   4b Search for a document by name, title, or author by selecting the appropriate radio button, entering identifying information, and clicking the **Search** button.



   This example shows a search for all resources that contain **PC** in their names.

   4c Select the document from the search results.

   Your choice is reflected under **Style Sheet File**.

4d Click **Close Window** to exit the Search For A Resource window.

5 To specify an external style sheet:

5a Click **Browse**.

A file selection dialog opens.

5b Browse to the appropriate style sheet and select it.

Your choice is reflected under **Style Sheet File**.

5c Next to **Style Sheet Folder**, click the ellipsis.

The Folder Selection dialog appears.

5d Navigate to the CMS Administration Console folder where you want to install the style sheet and click **Done**.

Your choice is reflected next to **Style Sheet Folder**.

6 Click **Save** to apply the style sheet specification to the document type properties.

➢ **To remove a style sheet specification from the document type properties:**

1 Enter templates mode by clicking the **Templates** button in the toolbar.

A panel appears listing all document types that have been defined.

2 Select the document type that contains the display style of interest.

A document type Property Inspector appears.

3 Click the **Advanced** tab.

4 Under Document Creation Style Sheet, click **Do Not Use Style Sheet**.

5 Click **Save** to remove the style sheet specification from the document type properties.

Creating taxonomies

If you plan to set up multiple categories for classifying documents, you may want to group them in a meaningful taxonomy.

📖 For more information, see "Classifying content" on page 25.

➢ **To create a taxonomy:**

1 Enter content mode by clicking the **Content** button in the toolbar.

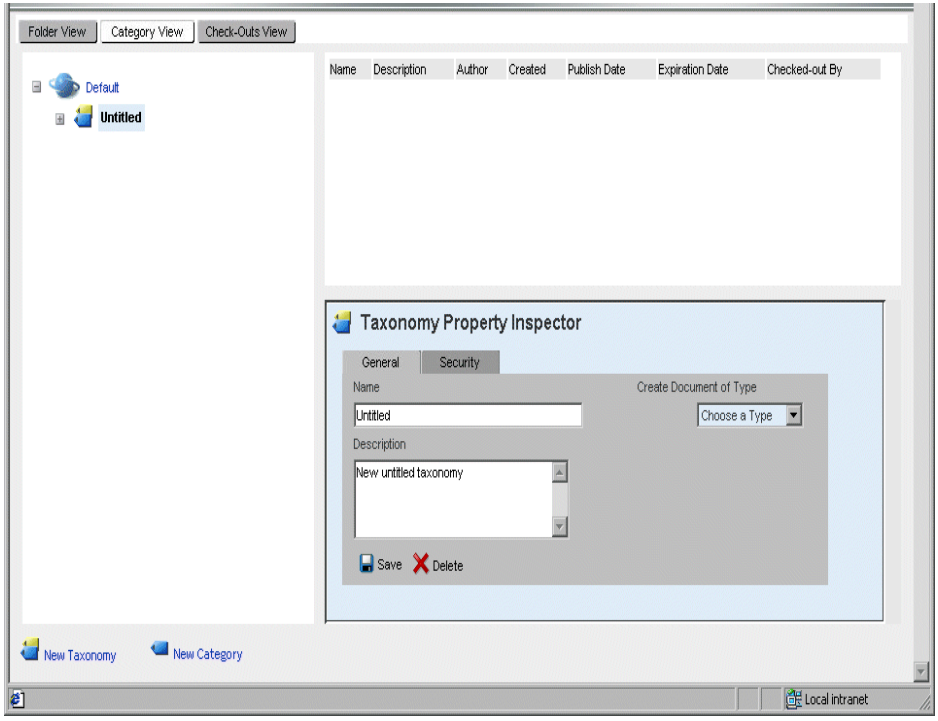
2 Select the **Category View** tab.

Your existing taxonomies and categories appear in the content tree view. (You may have to expand the Default root category.)

3 Click the **New Taxonomy** icon, located in the bottom-left panel of the CMS Administration Console.

An Untitled taxonomy appears in the content tree view.

4 Click **Untitled** to open the Property Inspector for the new taxonomy:



5 Fill in the **Name** and **Description** text boxes in the Property Inspector, then click **Save**.

The name of the taxonomy is updated in the content tree view.

6 Select the **Security** tab in the Property Inspector and set security for the taxonomy, as described in Chapter 19, "Managing Content Security".

7 Click **Save** to preserve your settings.

Creating categories

You can create one or more categories for classifying documents within a taxonomy.

For more information, see "Classifying content" on page 25.

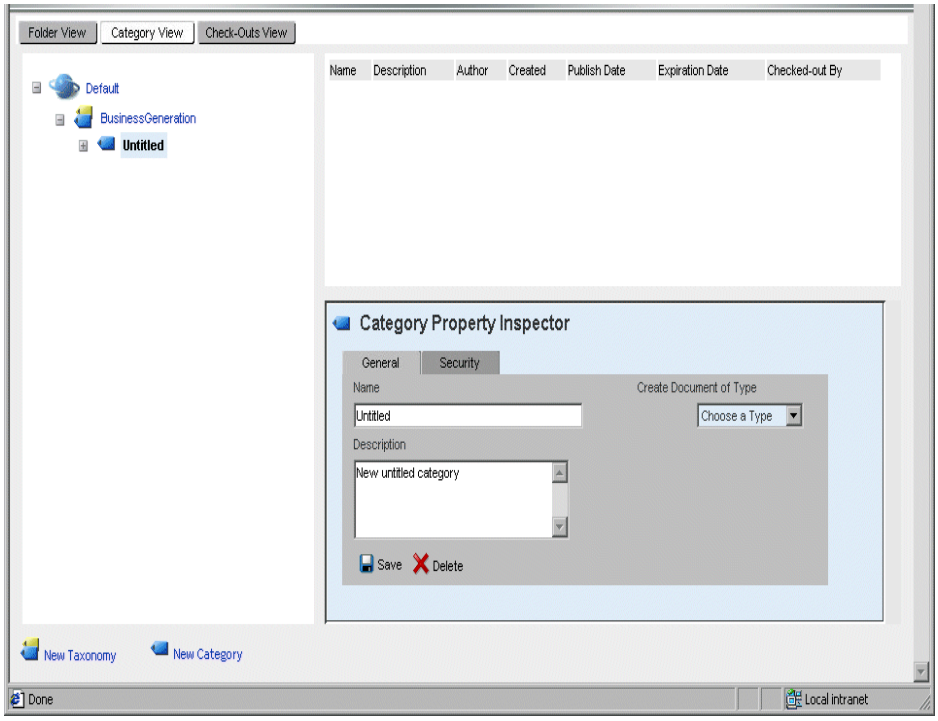
➢ **To create a category:**

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Category View** tab.

   Your existing taxonomies and categories appear in the content tree view.
3. Click the name of the taxonomy that will store your category.

   The name appears highlighted.
4. Click the **New Category** icon in the bottom-left panel of the CMS Administration Console.

   An Untitled category appears in the content tree view within the selected taxonomy.
5. Click **Untitled** to open the Property Inspector for the new category:



6. Fill in the **Name** and **Description** fields in the Property Inspector, then click **Save**.

   The name of the category is updated in the content tree view.
7. Select the **Security** tab in the Property Inspector and set security for the category, as described in Chapter 19, "Managing Content Security".
8. Click **Save** to preserve your settings.

15 Creating Content

This chapter describes how to create content using the CMS Administration Console. It has these sections:

- About content
- Flow of operations
- Creating documents
- Creating relationships between documents

About content

What content is *Content* is defined as information that is viewed or downloaded by users of your exteNd Director application. Content is managed in the CMS Administration Console. (It is important to distinguish content from *pages*, which are managed in the DAC and present the graphical interface that helps users navigate the Web site.)

📖 For more information about content, see Chapter 1, "About the Content Management Subsystem".

The CMS Administration Console supports content in any format that can be digitized, including HTML and binary content imported from other applications.

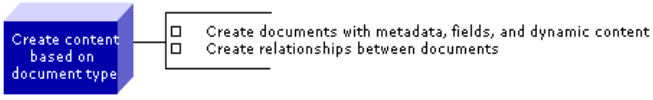
Before you create content Before you can create content for your exteNd Director application, the following elements of the content infrastructure must be in place:

Element	For information see
Folder for physically storing the content	"Creating folders" on page 198
Document type for defining content structure	"Creating document types" on page 199 and "Creating fields and adding them to a document type" on page 202

Within this infrastructure, you will be able to create content that conforms to the standards your organization has set for structure.

Flow of operations

Here is the basic task required to create content in the CMS Administration Console:



First you create content as documents based on a document type; then you can set up relationships between documents by adding child documents and attachments to a parent document. You can also set up relationships between documents by adding child documents and attachments to a parent document.

This section describes procedures for:

- Creating documents
- Creating relationships between documents

After the content has been developed, authorized users can add optional parts of the content infrastructure as needed—such as custom display styles, taxonomies, and categories. These procedures are covered in Chapter 14, "Setting Up the Optional Infrastructure".

Creating documents

With the CMS Administration Console, content developers create content in the form of *documents* that reside in folders. Each document is stored in one (and only one) folder.

When you create documents, you must specify three types of information:

1. Identifying information—or *metadata*:

Identifying information	Details
Name of document	Identifies the document in the content list)
Title of content	Appears in the user view of the document)
Subtitle	(Optional)
Author	—
Folder	Where document is stored)
Categories	(Optional)
Abstract	(Optional)
Status	(Optional)
Expiration date	(Optional)
Publish date	(Optional)

   - Name of document (identifies the document in the CMS Administration Console content list)
   - Title of content (appears in the user view of the document)
   - Subtitle (optional)
   - Author
   - Folder (where document is stored)
   - Categories (optional)
   - Abstract (optional)
   - Status (optional)
   - Expiration date (optional)
   - Publish date (optional)
2. Information required by the *fields* that are part of the document type
3. *Dynamic content* that can be entered either as HTML directly in the CMS Administration Console, or uploaded from external files

Each time you edit the content of a document, the CMS Administration Console creates a new version of the document content. The CMS Administration Console does not create a new version of the document content if you change only the metadata or custom field values but not the content.

If you want to create a document for the purpose of testing your style sheets, you can use the CMS Administration Console's Auto Create utility, which automatically fills in boilerplate content for you.

This section describes how to:

- Creating a document
- Specifying a folder for a new document
- Using Auto Create to create a document
- Using the CMS Administration Console's HTML Editor

Creating a document

➢ **To create a document:**

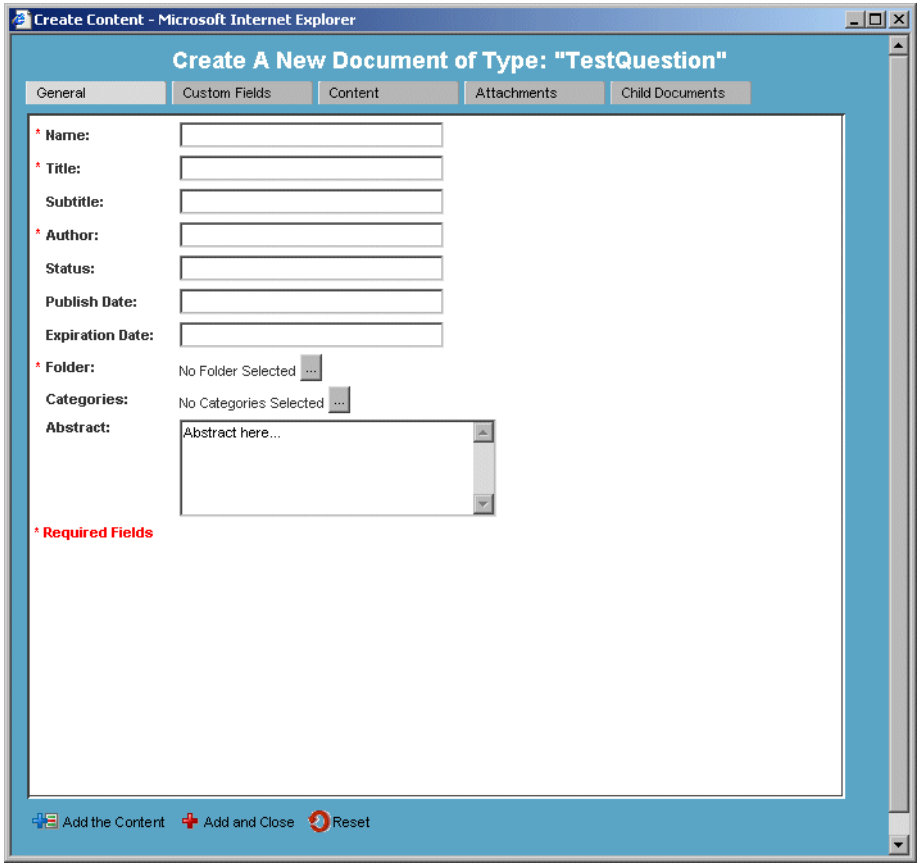
1. Enter templates mode by clicking the **Templates** button in the toolbar.

   A panel appears listing all document types that have been defined.

2 Select a document type from the list and click **Use**.

The Create A New Document window opens with the General tab open.

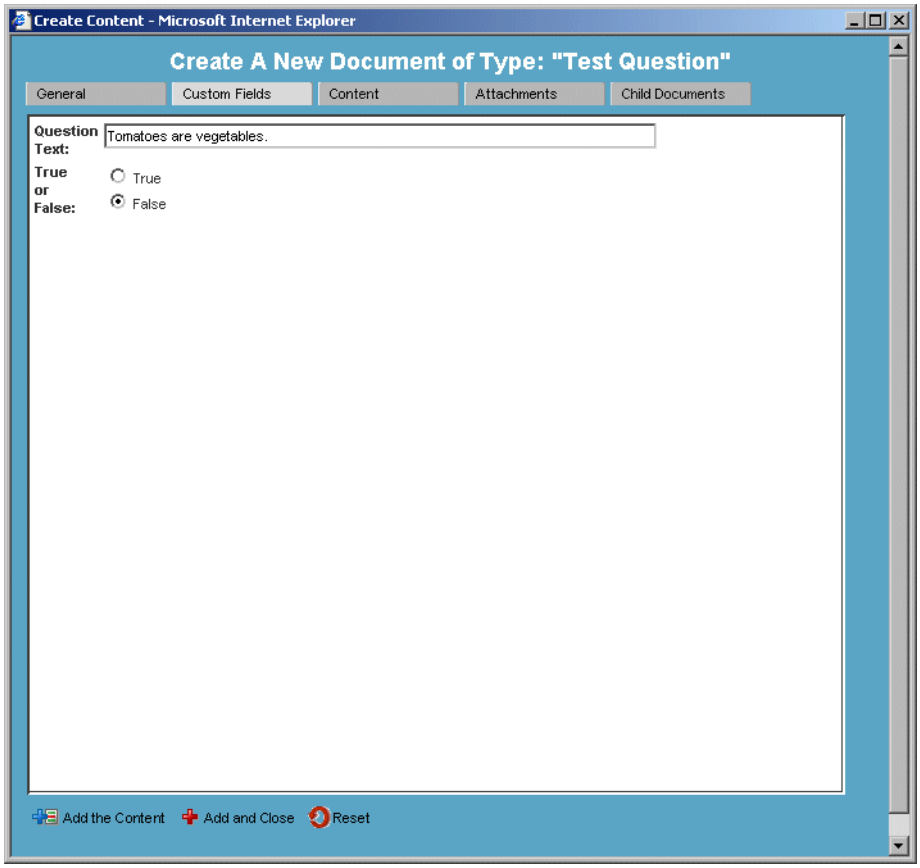
This tab contains the basic document metadata, such as name, title, author, folder that contains the document, any categories that contain the document, and so on:



3 Enter data into any required fields in the **General** tab.

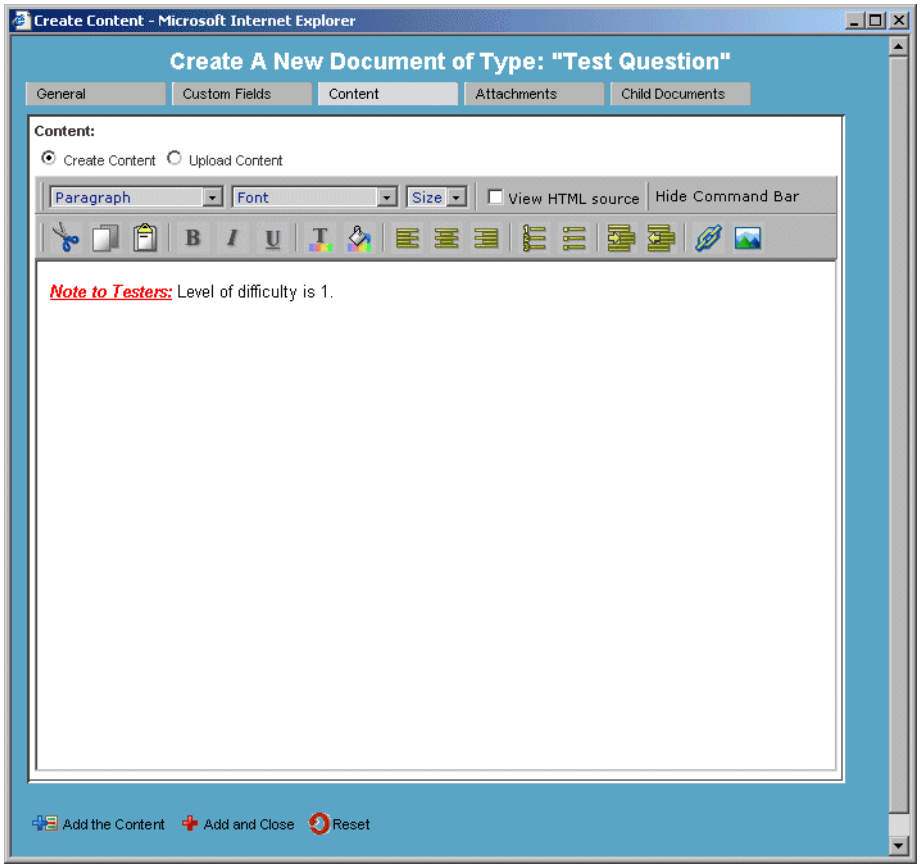
TIP: Any fields marked with an asterisk are required fields and must be filled in before you can create the document.

4 Click **Custom Fields** and enter data for any fields defined for the document type. Some example custom fields are shown below:



TIP: Custom fields are in some sense required fields in that you must fill in any empty fields before you can create the document. In the example above, you must fill in the Question Text field and select one of the buttons before the CMS Administration Console can create the document.

5 Click the **Content** tab and specify the dynamic content for the document:



The options for entering content depend on the Default Content setting of the document type (as specified under "Creating document types" on page 199):

- If Default Content = **Binary**, content developers upload content from an external file on the network.
- If Default Content = **HTML**, content developers use the CMS Administration Console's HTML Editor to enter content by typing directly in the edit area or by pasting in HTML source from an external editor.

- If Default Content = **Choice**, content developers can choose the way they enter content, as follows:

To	Do this
Create content in the HTML Editor	**1** Select the **Create Content** radio button. **2** Type or paste content in the CMS Administration Console's HTML Editor, using the command bar buttons to format text, create hyperlinks, and insert images.
Upload content from an external source	**1** Select the **Upload Content** radio button. **2** Enter a path to a file, or click **Browse** to navigate to a file on the network.

TIP: Users with appropriate privileges can modify the Default Content setting in the document type to restrict the type of content users can enter.

📖 To learn how to work with the CMS Administration Console's HTML Editor, see "Using the CMS Administration Console's HTML Editor" on page 229.

6 Click **Add the Content** at the bottom of the Create A New Document window.

The document is created in the folder you specified in the **General** tab.

📖 To view the content you just created, see "Previewing content" on page 242.

Specifying a folder for a new document

➢ **To specify a folder for a new document:**

1 Enter templates mode by clicking the **Templates** button in the toolbar.

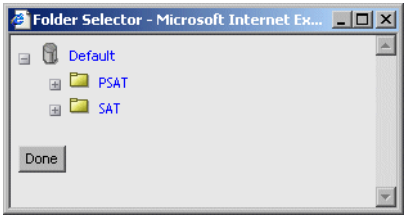
A panel appears listing all document types that have been defined.

2 Select a document type from the list and click **Use**.

The Create A New Document window opens.

3 Click the ellipsis next to the **Folder** field.

The Folder Selector window opens:



4 Navigate to the folder you want, click the folder name, and click **Done**.

The name of the selected folder appears in the Folder field of the Create Content window.

5 Enter other content as needed and click **Update the Content**.

Using Auto Create to create a document

➢ **To use Auto Create to create a document:**

1 Enter templates mode by clicking the **Templates** button in the toolbar.

A panel appears listing all document types that have been defined.

2 Select a document type from the list and click **Auto Create**.

The Create Content window opens, with most required metadata and fields filled in.

3 Specify a folder for the document.

4 Fill in any other content as desired and click **Add the Content**.

The document is created in the specified folder.

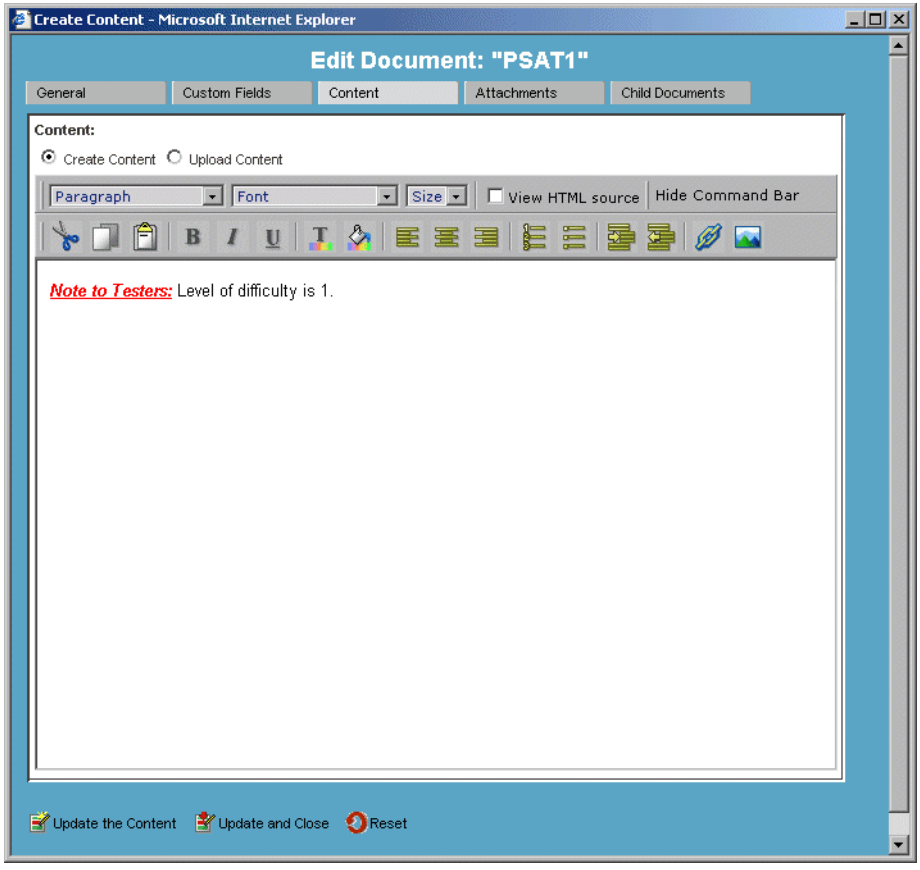
Using the CMS Administration Console's HTML Editor

Content developers can enter content as HTML using the CMS Administration Console's HTML Editor.

The only prerequisite is that you must set the Default Content option to **HTML** or **Choice** when creating the document type.

📖 For more information about specifying document type options, see "Creating document types" on page 199.

When you create or edit content using a document type with one of these Default Content settings (Binary, HTML, or Choice), the HTML Editor appears in the **Content** tab of the Create A New Document or Edit Document window:



With the CMS Administration Console's HTML Editor you can:

- Cut, copy, and paste text
- Format text
- Toggle editing mode between HTML code and rendered text
- Create hyperlinks
- Insert images

You can use the HTML Editor to edit the portion of the HTML code that would appear in the <BODY> section—not the entire HTML document. For example, you cannot use the HTML Editor to modify HTML code that would appear in the <HEAD> section of the document.

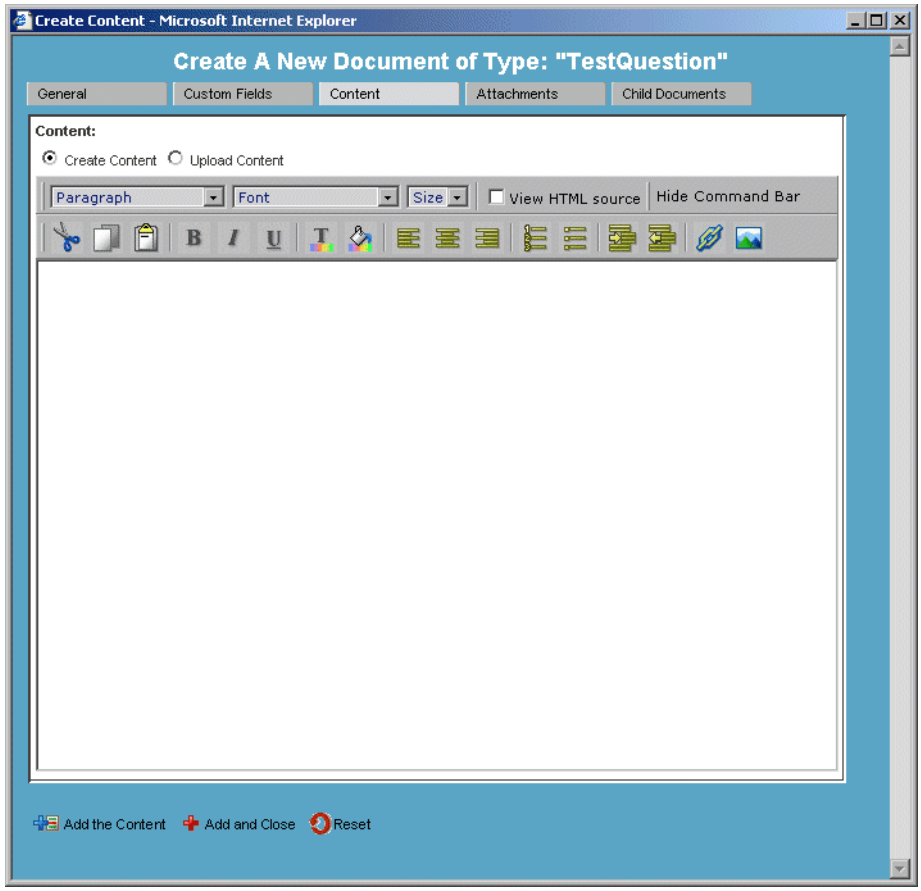
This section describes how to access and use the CMS Administration Console's HTML Editor.

➢ **To access the CMS Administration Console's HTML Editor when creating a new document:**

1. Enter templates mode by clicking the **Templates** button in the toolbar.

   A panel appears listing all document types that have been defined.
2. Select a document type whose Default Content field equals **HTML** or **Choice**.
3. Click **Use**.

   The Create A New Document window opens.
4. Click **Content** to access the HTML Editor:



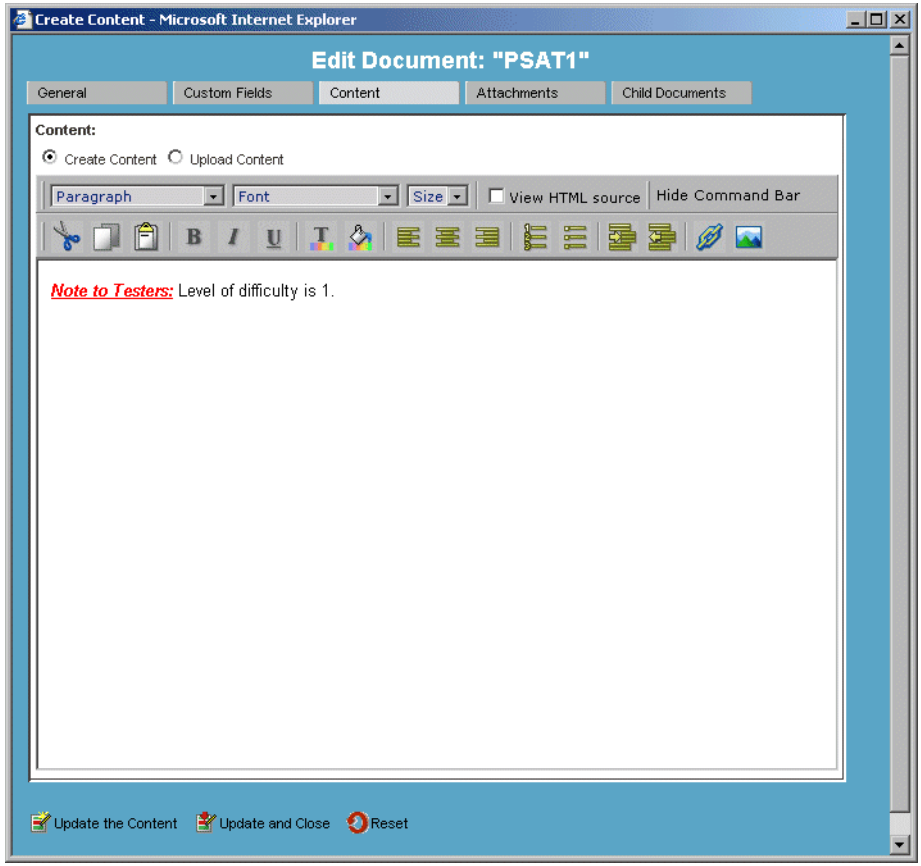
➢ **To access the CMS Administration Console's HTML Editor when editing an existing document:**

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your existing folders appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Navigate to the document of interest and select it to open its Property Inspector.
4. In the Property Inspector, select the **General** tab and click **Check-Out**.

   The CMS Administration Console checks out the latest version of the document.
5. Click the **Edit** button.

   The Edit Document window appears.
6. Click **Content** to access the HTML Editor:



For more information about checking documents in and out, see "Checking documents in and out" on page 260.

➢ **To cut, copy, paste, and format text:**

- Select the appropriate buttons from the toolbar.

➢ **To show HTML code:**

- Check **View HTML source**.
  Enabling this option:
  - Exposes HTML tags in existing text (entered while this setting was disabled)
  - Allows the HTML Editor to interpret HTML tags as code when typed in directly or pasted in from an outside source

    NOTE: If you enter HTML tags when this option is disabled, the tags are not interpreted as code and instead are converted to text.

➢ **To show rendered text:**

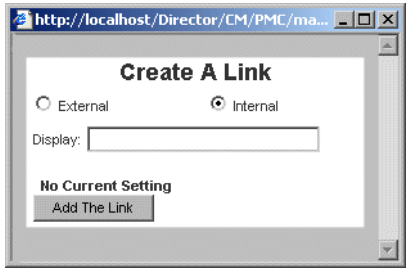
- Deselect **View HTML source** to hide HTML tags and show rendered text.

➢ **To create a hyperlink:**

1. Position the cursor in the HTML Editor where you want to insert the link.
2. Click the **Create Hyperlink** button:

   The Create A Link window opens:

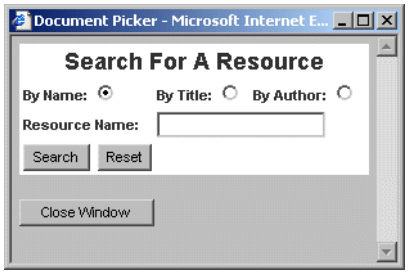
3. Choose the type of link you want to create:

Type of link	Description
Internal	Link to content that you created in or uploaded to the CMS Administration Console
External	Link to external content

4 To create an **internal** link:

4a Select the **Internal** radio button.

The Search For A Resource window opens:



4b Search for internal content by name, title, or author by selecting the appropriate radio button, entering the appropriate identifying information, and clicking the **Search** button.

4c Select the resource from the search results and click the **Close Window** button at the bottom of the window.

A text string linking to the resource appears in the Create A Link window. You can click on the text string to view the resource.

4d Back in the Create A Link window, enter the display text for the link in the **Display** field and click **Add The Link**.

5 To create an **external** link:

5a Select the **External** radio button.

5b Enter the display text for the link in the **Display** field.

5c Enter the URL of the external content in the **URL** field.

NOTE: You can enter an URL that invokes a servlet to serve up content to your exteNd Director application.

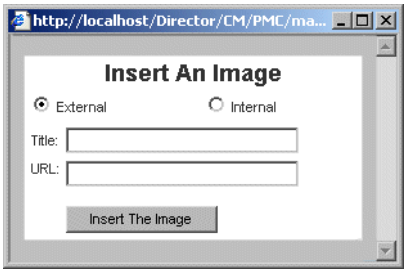
5d Click **Add The Link**.

➢ **To insert an image:**

1 Position the cursor in the HTML Editor area where you want to insert the image.

2 Click the **Insert Image** button:

The Insert An Image window opens:



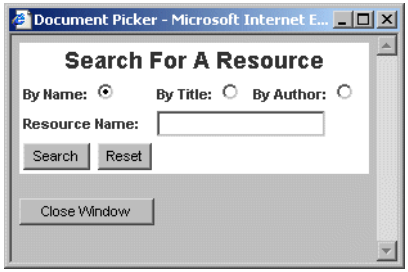
3 Choose the type of image you want to insert:

Type of image	Description
Internal	Image that you created in or uploaded to the CMS Administration Console
External	Image created outside the CMS Administration Console

4 To insert an **internal** image:

4a Select the **Internal** radio button.

The Search For A Resource window opens:



4b Search for an internal image by name, title, or author by selecting the appropriate radio button, entering the appropriate identifying information, and clicking the **Search** button.

4c Select the image from the search results and click the **Close Window** button at the bottom of the window.

A text string identifying the image target appears in the Insert An Image window.

4d Back in the Insert An Image window, enter a title for the image in the **Title** field.

The title is the hover text that will appear as the cursor moves over the image.

4e Click **Insert The Image**.

5 To insert an **external** image:

5a Select the **External** radio button.

5b Enter a title for the image in the **Title** field.

The title is the hover text that will appear as the cursor moves over the image.

5c Enter the URL of the external image in the **URL** field.

5d Click **Insert The Image**.

Creating relationships between documents

The CMS Administration Console allows you to create two types of relationships between documents:

Relationship type	Description
Parent/child	In this relationship, a parent document can have one or more child documents. This is a *one-to-many relationship*: each child document can have only one parent, but each parent can have multiple child documents. A typical application of the parent/child relationship is for a discussion thread in which one question can have multiple responses but each response relates to only one parent question.
Parent/attachment	In this relationship, a parent document can have one or more attached documents. This is a *many-to-many relationship*: each parent document can have more than one attachment, while each attachment can be shared with multiple other parents. A typical application of the parent/attachment relationship is an online bookstore that attaches author documents to its book lists, where multiple books can have the same author.

The definition of *document* includes not only documents created in the CMS Administration Console, but also documents that are uploaded to the CMS Administration Console, such as images and binary files.

This section describes how to add child documents and attachments to a parent document, and how to remove these relationships.

➢ **To add a child document:**

Users with READ and WRITE permissions can add children to a document. You can add internal child documents or upload external documents.

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Click to select the folder that contains the content of interest.

   A list of documents appears in the content list.
4. Select the document of interest.

   A **Child Docs** tab appears in the document's Property Inspector.
5. Click **Check-Out** to check out your document and then select the **Child Docs** tab.
6. Select a document, using one of these methods:

To	Do this
Add an internal document	1 Click **Add** in the Child Docs pane. The Search For A Resource window opens. 2 Search for a document by name, title, or author by selecting the appropriate radio button, entering identifying information, and clicking the **Search** button. 3 Select the document from the search results.
Upload an external document	1 Click **Upload** in the Child Docs pane. The Upload A File Attachment window opens. 2 Browse to the document of interest and select it. 3 Click **Upload**.

   The document you select appears as a child of your document in the Property Inspector.
7. Check your document back in by selecting the **General** tab, then clicking **Check-In**.

📖 For more information about checking documents in and out, see "Checking documents in and out" on page 260.

➢ **To add an attachment:**

Users with READ and WRITE permissions can add attachments to a document. You can attach internal documents or upload external documents.

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Click to select the folder that contains the content of interest.

   A list of documents appears in the content list.
4. Select the document of interest.

   An Attachments tab appears in the document's Property Inspector.
5. Click **Check-Out** to check out your document and then select the **Attachments** tab.
6. To attach an **internal** document (one that has been created in or uploaded to the CMS Administration Console):
   - **6a** Click **Add** in the Attachments pane.

     The Search For A Resource window opens.
   - **6b** Search for a document by name, title, or author by selecting the appropriate radio button, entering identifying information, and clicking the **Search** button.
   - **6c** Select the document from the search results.

     The Attachment Properties window opens.
   - **6d** (Optional) In the Description text area, enter text about the relationship between the parent document and its attachment.

     This text appears in the XML generated by the CMS Administration Console Content Reader.
   - **6e** Click **Add**.

     The document you selected appears as an attachment to your document in the Property Inspector.
7. To attach an **external** document:
   - **7a** Click **Upload** in the Child Docs pane.

     The Upload A File Attachment window opens.
   - **7b** Browse to the document of interest and select it.
   - **7c** (Optional) In the Description text area, enter text about the relationship between the parent document and its attachment.

     This text appears in the XML generated by the CMS Administration Console Content Reader.

7d Click **Upload**.

The document you select appears as an attachment to your document in the Property Inspector.

8 Check your document back in by selecting the **General** tab, then clicking **Check-In**.

📖 For more information about checking documents in and out, see "Checking documents in and out" on page 260.

➢ **To remove relationships between documents:**

To remove the relationship between a parent document and its child or attachment, you need READ, WRITE, and LIST permissions.

1 Enter content mode by clicking the **Content** button in the toolbar.

2 Select the **Folder View** tab.

Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.

3 Click to select the folder that contains the content of interest.

A list of documents appears in the content list.

4 Select the parent document of interest to open its Property Inspector.

5 Check out the parent document by clicking **Check-Out** in its Property Inspector.

6 Check out the attachment or child document of interest by selecting it in the content list and clicking **Check-Out** in its Property Inspector.

7 Select the parent document again and then choose the **Attachments** or **Child Docs** tab in its Property Inspector.

8 Select the attachment or child document of interest in the parent's Property Inspector.

The Property Inspector refreshes to provide a Remove Relationship button.

9 Click the **Remove Relationship** button.

The attachment or child document disappears from the parent's Property Inspector, but remains in its CMS Administration Console folder.

10 Check the parent and child (or attachment) back in by selecting the **General** tab, then clicking **Check-In**.

NOTE: The parent and child (or attachment) must both be checked out to sever the relationship. Otherwise, the **Remove Relationship** button will not appear. Even after you sever the relationship, the attached file or child document remains in the CMS Administration Console.

📖 For more information about checking documents in and out, see "Checking documents in and out" on page 260.

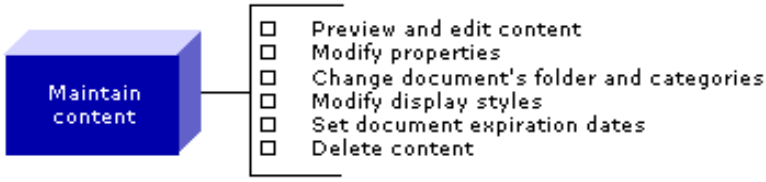
16 Maintaining Content

This chapter describes various ways to access and update existing content using the CMS Administration Console. It has these sections:

- Flow of operations
- Previewing content
- Editing content
- Modifying properties
- Assigning a document's folder, categories, and taxonomies
- Modifying display styles
- Editing document types
- Editing document fields
- Setting document expiration dates
- Deleting content

Flow of operations

Here is a workflow that shows the variety of operations available to authorized users who are responsible for maintaining content in the CMS Administration Console:



This section presents procedures for:

- Previewing content
- Editing content
- Modifying properties
- Assigning a document's folder, categories, and taxonomies
- Modifying display styles
- Editing document types
- Editing document fields
- Setting document expiration dates
- Deleting content

Previewing content

Users with READ permission can preview documents to get a view of how content will appear to users of the Web site. Using the preview function, document reviewers verify the accuracy, structure, and layout of content before it is published.

➢ **To preview the latest version of content:**

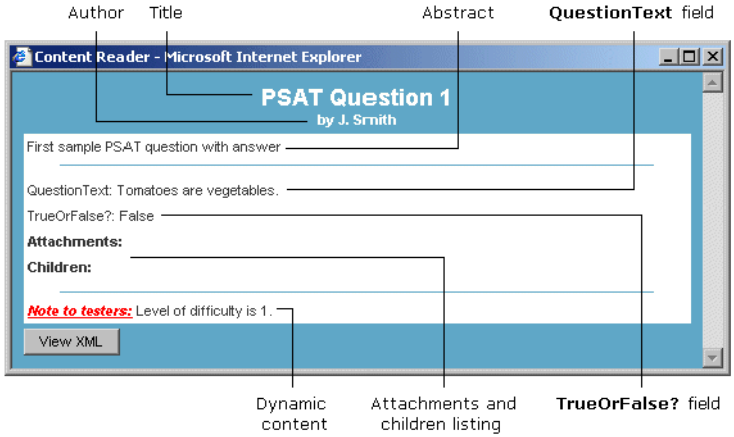
1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Click to select the folder that contains the content of interest.

   A list of documents appears in the content list.
4. Select the document of interest to open its Property Inspector.

5 Click the **Preview** button.

The latest version of the document's content opens in the Content Reader window:



➢ **To preview a specific version of content:**

1 Enter content mode by clicking the **Content** button in the toolbar.

2 Select the **Folder View** tab.

Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.

3 Click to select the folder that contains the content of interest.

A list of documents appears in the content list.

4 Select the document of interest to open its Property Inspector.

5 Select the **Versions** tab.

A list of content versions appears, ordered from most recent to earliest.

The currently published version of content appears with the published-version icon:

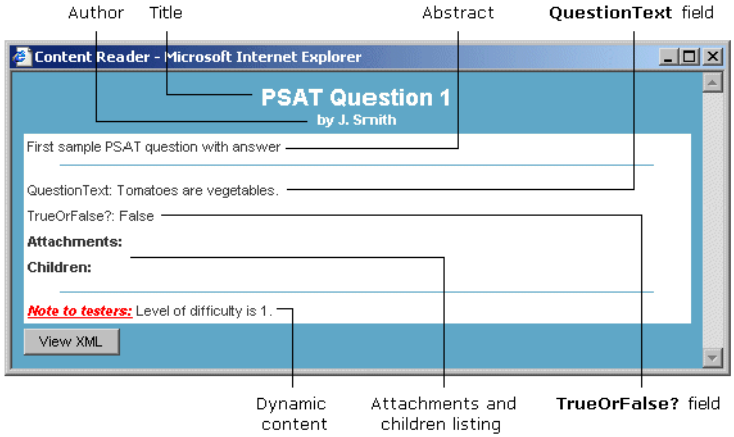
If no version has been published, all versions appear with the default document icon:

6 Click to select a version.

The version name appears highlighted.

7 Click the **Preview** button.

The selected version of the document's content opens in the Content Reader window:



NOTE: If no version of this document has been published, no dynamic content appears in the Content Reader. Instead, a message appears at the bottom of the Content Reader window indicating that there is no currently published content for the document. For information about publishing content, see "Administering version control" on page 264.

Editing content

Users with READ and WRITE permission can edit content. Documents must be checked out before they can be modified. The CMS Administration Console applies edits to the latest version of a document and saves the modifications as a new (later) version.

➢ **To edit content:**

1 Enter content mode by clicking the **Content** button in the toolbar.

2 Select the **Folder View** tab.

Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.

3 Click to select the folder that contains the content of interest.

A list of documents appears in the content list.

4 Select the document of interest to open its Property Inspector.

5 Click the **Check-Out** button.

6 Click the **Edit** button.

An edit window appears in which metadata, fields, and dynamic content can be modified.

7 Edit the content, then click **Update The Content**.

NOTE: To undo your edits, click the **Reset** button to return the document to its original state.

The updated content is saved in a new version of the document.

8 Check the document back in by clicking **Check-In**.

📖 For more information about checking documents in and out, see "Checking documents in and out" on page 260.

Modifying properties

Users with READ, WRITE, and LIST permissions can modify the properties of the following CM elements in the CMS Administration Console:

- Folders
- Taxonomies
- Categories
- Documents
- Values of document fields

➢ **To modify properties:**

1 Select the CM element of interest and open its Property Inspector.

Here's how to access the Property Inspector for each element:

CM element	How to access
Folder	1 Click the **Content** button. 2 Select the **Folder View** tab. 3 Select the folder of interest.
Taxonomy and category	1 Click the **Content** button. 2 Select the **Category View** tab. 3 Select the taxonomy or category of interest.

CM element	How to access
Document	**1** Click the **Content** button. **2** Select the **Folder View** tab. **3** Expand the folder that contains the document of interest. **4** Select the document. **5** Check out the document by clicking **Check-Out**.
Document field	**1** Click the **Templates** button. **2** Select a document type that contains the field of interest. **3** Select the field and click the **Properties** button.

2 In the Property Inspector, modify properties as needed.

TIP: Some properties cannot be edited.

3 Record your changes:

For	Do this
Folders, taxonomies, categories, and documents	Click **Save**.
Document fields	**1** Click **Update**. **2** Check the document back in by clicking **Check-In**.

For more information about checking documents out and in, see "Checking documents in and out" on page 260.

Assigning a document's folder, categories, and taxonomies

You can change the folder, categories, and taxonomies anytime for any document for which you have READ, WRITE, and LIST permissions.

➢ **To change a document's folder:**

1 Enter content mode by clicking the **Content** button in the toolbar.

2 Select the **Folder View** tab.

Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.

3 Click the folder that contains the document of interest.

A list of documents appears in the content list.

4 Select the document of interest to open its Property Inspector.

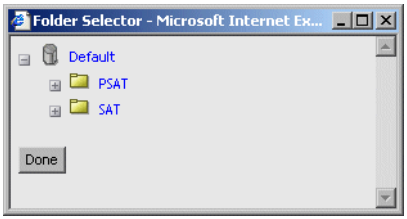
5 Click the **Check-Out** button.

6 Click the **Edit** button.

An edit window appears.

7 Click the ellipsis next to the **Folder** field.

The Folder Selector window opens:



8 Navigate to the new folder, click the folder name, and click **Done**.

The name of the new folder replaces the old one in the Folder field of the edit window.

9 Click **Update The Content**.

10 Click **Check-In**.

➢ **To assign a document to categories or taxonomies:**

1 Enter content mode by clicking the **Content** button in the toolbar.

2 Select the **Folder View** tab.

Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.

3 Click the folder that contains the document of interest.

A list of documents appears in the content list.

4 Select the document of interest to open its Property Inspector.

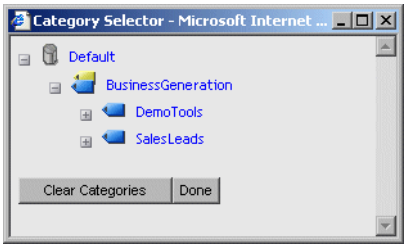
5 Click the **Check-Out** button.

6 Click the **Edit** button.

An edit window appears.

7 Click the ellipsis next to the **Categories** field.

The Category Selector window opens:



8 Navigate to an appropriate category or taxonomy and click the name.

The name of the new category appears in the **Categories** field of the edit window.

You can click additional categories and taxonomies to add the document to them.

TIP: If you click a category or taxonomy that already contains the document, that document is removed from that category or taxonomy. (In the Edit Document dialog, the document's name is removed from the **Categories** listing.)

9 When you have finished specifying categories and taxonomies, click **Done**.

The Category Selector window closes and your choices are reflected in the **Categories** listing.

10 Enter other content as needed and click **Update And Close**.

The Edit Document dialog closes.

11 In the Content Property Inspector, click **Check-In**.

➢ **To change a document's categories or taxonomies:**

1 Enter content mode by clicking the **Content** button in the toolbar.

2 Select the **Category View** tab.

Your taxonomies and categories appear in the content tree view. You may need to expand some of these containers to see the complete view.

3 Click the category or taxonomy that contains the document of interest.

A list of documents appears in the content list.

4 Select the document of interest to open its Property Inspector.

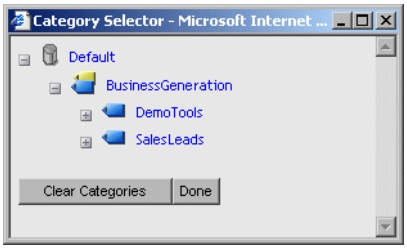
5 Click **Check-Out**.

6 Click **Edit**.

An edit window appears.

7 Select the ellipsis next to the **Categories** field.

The Category Selector window opens:



8 Navigate to the appropriate category or taxonomy and click the name.

The name of the new category appears in the **Categories** field of the edit window.

You can click additional categories and taxonomies to add the document to them.

To remove the document from a category or taxonomy, click that category or taxonomy. (In the Edit Document dialog, the document's name is removed from the **Categories** listing.)

9 When you have finished specifying categories and taxonomies, click **Done**.

The Category Selector window closes and your choices are reflected in the **Categories** listing.

10 Click **Update The Content**.

11 Click **Check-In**.

Modifying display styles

Authorized users can modify a display style by uploading changes to its XSL style sheets. The CMS Administration Console stores these updates as new versions of the style sheets. Users then publish the version they want to apply to content.

This section describes the procedure for modifying style sheets in a display style.

NOTE: Before you begin, make sure you have updated the style sheet in an external editor and can access the file containing these modifications from your local file system, the network, or the CMS Administration Console.

➢ **To modify a display style:**

1 Enter content mode by clicking the **Content** button in the toolbar.

2 Select the **Folder View** tab.

Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.

3 Navigate to the folder that contains the XSL style sheet you want to modify.

TIP: Style sheets appear as system resources.

4 Select the style sheet of interest to open its Property Inspector.

5 Click the **Check-Out** button.

The style sheet is checked out and appears with the checked-out document icon:

6 Switch to templates mode by clicking the **Templates** button in the toolbar.

A panel appears listing all document types that have been defined.

7 Select the document type that contains the display style you want to change.

A document type Property Inspector appears.

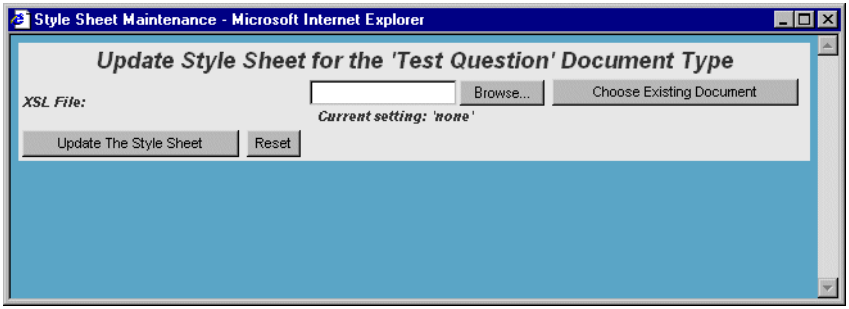
8 Click the **XSL Style Sheets** tab.

Two panes appear. The Default Display Styles pane lists the display styles that have been created for the document type, and the Properties pane displays the properties of a selected display style.

9 In the Default Display Styles pane, expand the display style you want to modify to display its associated XSL style sheets.

10 Select the style sheet you want to modify and click **Upload**.

The Update Style Sheet window opens:



11 Enter the name of the updated XSL style sheet using one of these methods:

- Browse the network for an external file.

  OR

- Select **Choose Existing Document** to search for an updated XSL file that has already been uploaded to the CMS Administration Console.

A new version of the XSL style sheet is created.

12 Enter content mode by clicking the **Content** button in the toolbar.

The style sheet document should still be selected with its Property Inspector open.

13 Check the style sheet back in by clicking **Check-In**.

For more information about checking documents in and out, see "Checking documents in and out" on page 260.

NOTE: To apply the updated style sheet to content, you must publish the new version, as described in "Administering version control" on page 264.

Editing document types

Authorized users can edit document types. All changes apply to legacy documents as well as new documents of the designated type.

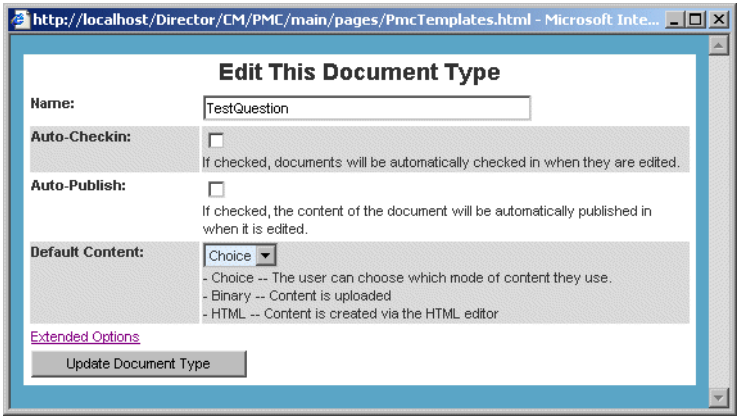
➢ **To edit a document type:**

1 Enter templates mode by clicking the **Templates** button in the toolbar.

A panel appears listing the document types that have been defined.

2 Select the document type you want to modify and click **Edit**.

The Edit This Document Type window opens:



NOTE: If you created the document outside of the CMS Administration Console using the CM APIs, you might not be able to access the document type and associated data. The CMS Administration Console requires that certain meta data be included. For more information, refer to EbiContentMgmtDelegate in the on-line *API Reference* section.

3 (Optional) Click **Extended Options** to display additional document type options.

4 Edit fields and options as needed.

For details about the individual options, see "Creating document types" on page 199.

5 Click **Update Document Type**.

Editing document fields

Authorized users can edit fields, but only from within the document types where the fields were originally created.

➢ **To edit a document field:**

1. Enter templates mode by clicking the **Templates** button in the toolbar.

   A panel appears listing all document types that have been defined.

2. Select the document type in which the field was created.

   The fields defined for that document type appear along with the list of all available fields.

   NOTE: If you created the document outside of the CMS Administration Console using the CM APIs, you might not be able to access the document fields. The CMS Administration Console requires that meta data from the document fields be included. For more information, refer to EbiContentMgmtDelegate in the on-line *API Reference* section.

   The Available Fields list displays the parent document type in parentheses next to each field. Use this information to verify that you are editing the field in its parent document type.

3. Select the field you want to edit and click **Properties**.

   The Property Inspector opens.

4. Edit the properties of the field as appropriate and click **Update**.

Setting document expiration dates

There are occasions when a content administrator needs to set an expiration date for a documents that has a limited life span. The CMS Administration Console allows users with WRITE permission to set or change this date anytime after the document is created.

When expiration dates are set, developers can write queries in portlets to remove expired content, or write a scheduled business object to check expiration dates and take specified actions if content is obsolete.

➢ **To set the expiration date of a document:**

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.

3 Click to select the folder that contains the document of interest.

A list of documents appears in the content list.

4 Select the document of interest to open its Property Inspector.

5 Click the **Check-Out** button.

6 In the **Expiration Date** field, enter an expiration date of the form:

```
YYYY-MM-DD HH:MM:SS
```

7 Click **Save**.

8 Click the **Check-In** button.

📖 For more information about checking documents in and out, see "Checking documents in and out" on page 260.

Deleting content

Authorized users can delete certain CM elements in the CMS Administration Console. This section describes procedures for:

- Deleting folders
- Deleting taxonomies and categories
- Deleting documents
- Deleting display styles
- Deleting document types
- Deleting and removing document fields

Deleting folders

When you delete a folder, all folders and documents it contains are also deleted.

➢ **To delete a folder:**

1 Enter content mode by clicking the **Content** button in the toolbar.

2 Select the **Folder View** tab.

Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.

3 Select the folder of interest to open its Property Inspector.

4 Click **Delete**.

5 When a confirmation window appears, click **OK**.

Deleting taxonomies and categories

When you delete a taxonomy or category, all categories it contains are also deleted. Documents are always retained in their parent folder, even if their assigned taxonomies or categories have been removed.

➢ **To delete a taxonomy or category:**

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Category View** tab.

   Your taxonomies and categories appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Select the taxonomy or category of interest to open its Property Inspector.
4. Click **Delete**.
5. When a confirmation window appears, click **OK**.

Deleting documents

You must check out a document before you can delete it. When you delete a document, all versions are removed.

➢ **To delete a document:**

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Navigate to the document of interest and select it to open its Property Inspector.
4. In the Property Inspector, select the **General** tab and click **Check-Out**.

   The Property Inspector refreshes to display new function buttons.
5. Click **Delete**.
6. When a confirmation window appears, click **OK**.

📖 For more information about checking documents in and out, see "Checking documents in and out" on page 260.

Deleting display styles

When you delete a display style, the CMS Administration Console also removes all XSL style sheets that have been created for that display style.

➢ **To delete a display style:**

1. Enter templates mode by clicking the **Templates** button in the toolbar.

   A panel appears listing all document types that have been defined.
2. Select the document type that contains the display style to delete.
3. Select the **XSL Style Sheets** tab.

   A list of the document type's display styles appears in the Default Display Styles pane.
4. Select the display style you want to delete.
5. Click **Delete** under the Properties pane.
6. When a confirmation window appears, click **OK**.

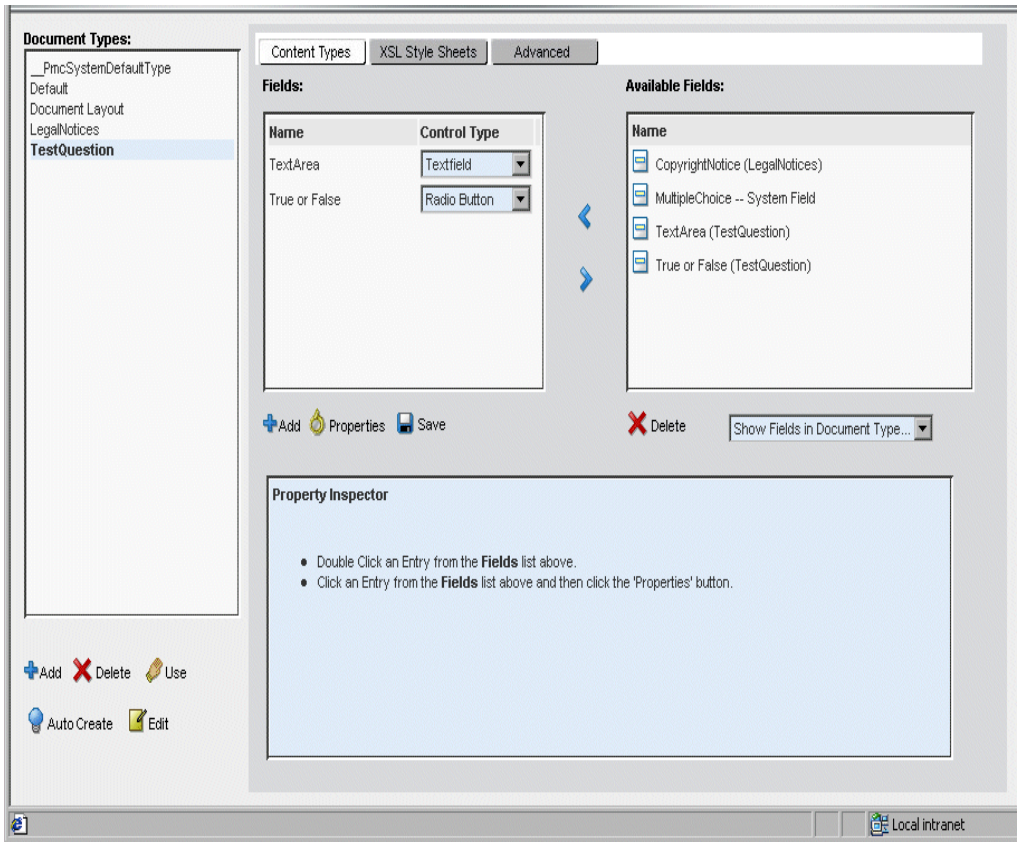
Deleting document types

When you delete a document type, the CMS Administration Console also removes all documents that have been created using that document type.

There is another side effect of deleting document types: any fields that were created within that document type are adopted by a new parent—the *system document type*—that appears in the Document Types list as **_PmcSystemDefaultType**.

Once adopted, these fields remain part of the available pool of fields but can be edited only from within **_PmcSystemDefaultType**. You can easily identify adopted system fields: they appear in the Available Fields pool with the suffix **--System Field** appended to their names.

In the following example, **Multiple Choice** is an adopted system field:



➢ **To delete a document type:**

1. Enter templates mode by clicking the **Templates** button in the toolbar.

   A panel appears listing all document types that have been defined.
2. Select the document type to delete.
3. Click **Delete** under the Document Types pane.
4. When a confirmation window appears, click **OK**.

   The CMS Administration Console deletes the document type and all documents that have been created using that document type.

Deleting and removing document fields

There are two separate operations:

Operation	Description
Permanently deleting fields—from the CMS Administration Console	Deletes fields from all documents and from pool or available fields
Removing fields—from the parent document type	Removed fields from all documents—but leaves fields in pool of available fields

Permanently deleting fields—from the CMS Administration Console

You can delete document fields permanently from the CMS Administration Console, but only from within the document types where they were originally defined. When you delete a field from the CMS Administration Console, the field is removed from all existing documents in which it appeared and from the pool of available fields.

CAUTION: *Although this is a convenient way of applying one deletion to multiple documents, be aware that the effect is global and irreversible.*

➢ **To permanently delete a document field from the CMS Administration Console:**

1. Enter templates mode by clicking the **Templates** button in the toolbar.

   A panel appears listing the document types that have been defined.
2. Select the document type for which the field was defined.
3. Select the field in the Available Fields pane.
4. Click **Delete** under the Available Fields pane.
5. When a confirmation window appears, click **OK**.

   The CMS Administration Console deletes the field from the Available Fields pane and from all documents that have been created using document types that contain the field.

Removing fields—from the parent document type

You can remove a document field from the document types where it was originally defined but leave it in the available pool of fields for later use.

When a field is removed from its parent document type, it is adopted by the system document type **_PmcSystemDefaultType**. You can then add the field to any document type, but edit it only from the system document type.

What happens to legacy documents when you remove a field from its parent document type? There are two scenarios:

If you	The CMS Administration Console
Selected the Clean Up Data option in the parent document type	Deletes the field from legacy documents of the designated type
Did **not** select the Clean Up Data option in the parent document type	Preserves the field in legacy documents of the designated type, but does not allow you to edit the field **NOTE:** You will see the legacy field when you preview the document, but not when you edit the document.

For more information about the Clean Up Data option, see "Creating document types" on page 199.

➢ **To remove a document field from a document type (but leave it available):**

1. Enter templates mode by clicking the **Templates** button in the toolbar.

   A panel appears listing the document types that have been defined.
2. Select the document type for which the field was defined.
3. Select the field in the Fields pane and then click the **Remove Field** button:

   The field is removed from the Fields pane and refreshes in the Available Fields pane as a system field.
4. Click **Save**.

17 Administering Content

This chapter describes how to check documents in and out and administer version control. It has these sections:

- About content administration
- Flow of operations
- Checking documents in and out
- Administering version control

About content administration

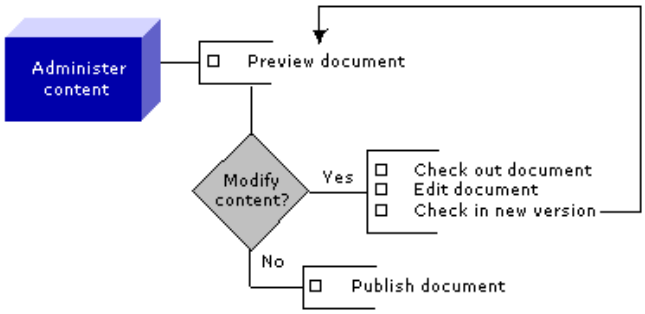
In organizations responsible for developing and maintaining exteNd Director applications, CM is a dynamic process that often involves multiple users interacting concurrently with a shared set of files within a common infrastructure.

To preserve the integrity of data in this type of environment, the CMS Administration Console provides a number of safeguards for effectively administering content:

- Ability to lock documents using checkin and checkout functions
- Version control

Flow of operations

Here is a workflow that illustrates the recommended order of operations for administering content in the CMS Administration Console:



Checking documents in and out

To prevent concurrent access to documents in a multiuser environment, the CMS Administration Console provides **checkin** and **checkout** capability to users with READ, WRITE, and LIST permissions—typically the users who are content developers and administrators.

Authorized users must check out documents before they can make any changes to the content, including:

- Modifying properties
- Changing field values
- Updating HTML content
- Adding child documents
- Adding attachments

These rules also apply to XSL style sheets, which when uploaded to the CMS Administration Console are managed in the same way as documents that are created in the CMS Administration Console.

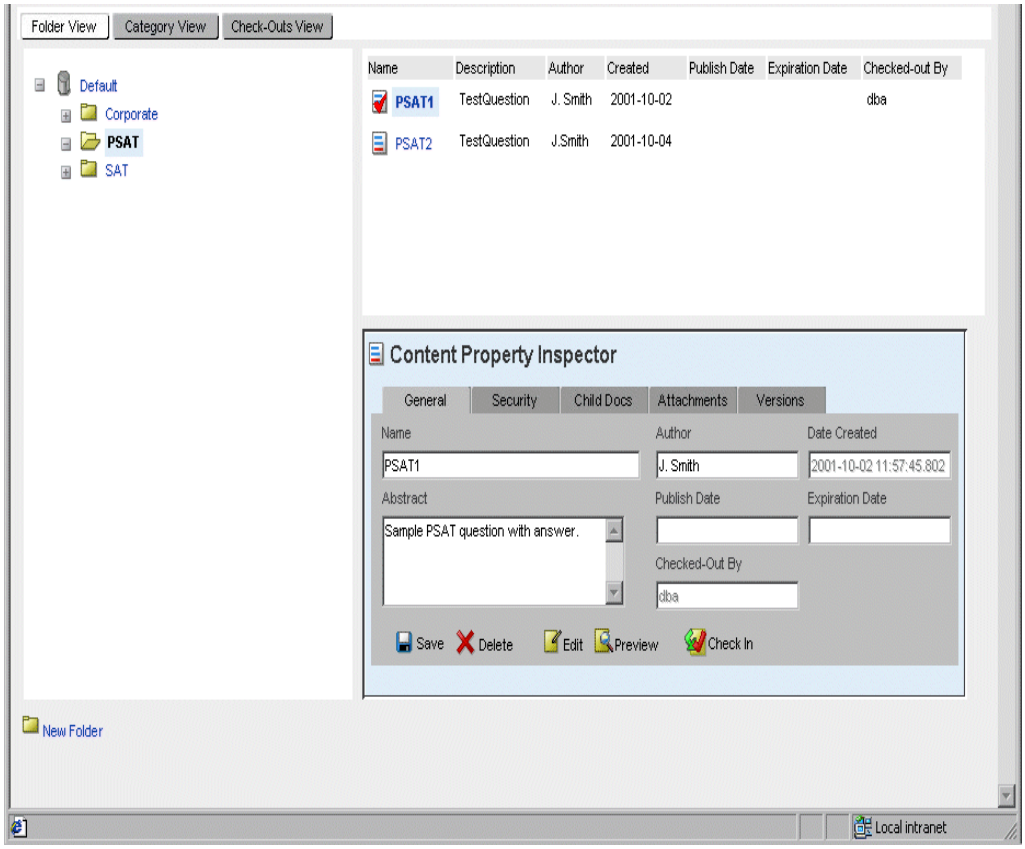
This section describes what happens during checkin and checkout and explains how to perform the following tasks:

- Check out a document
- Check in a document
- Enable automatic checkin

What happens during checkout

Checking out a document locks it, preventing other users from modifying the content. Users with READ permission can view the currently published content of checked-out documents.

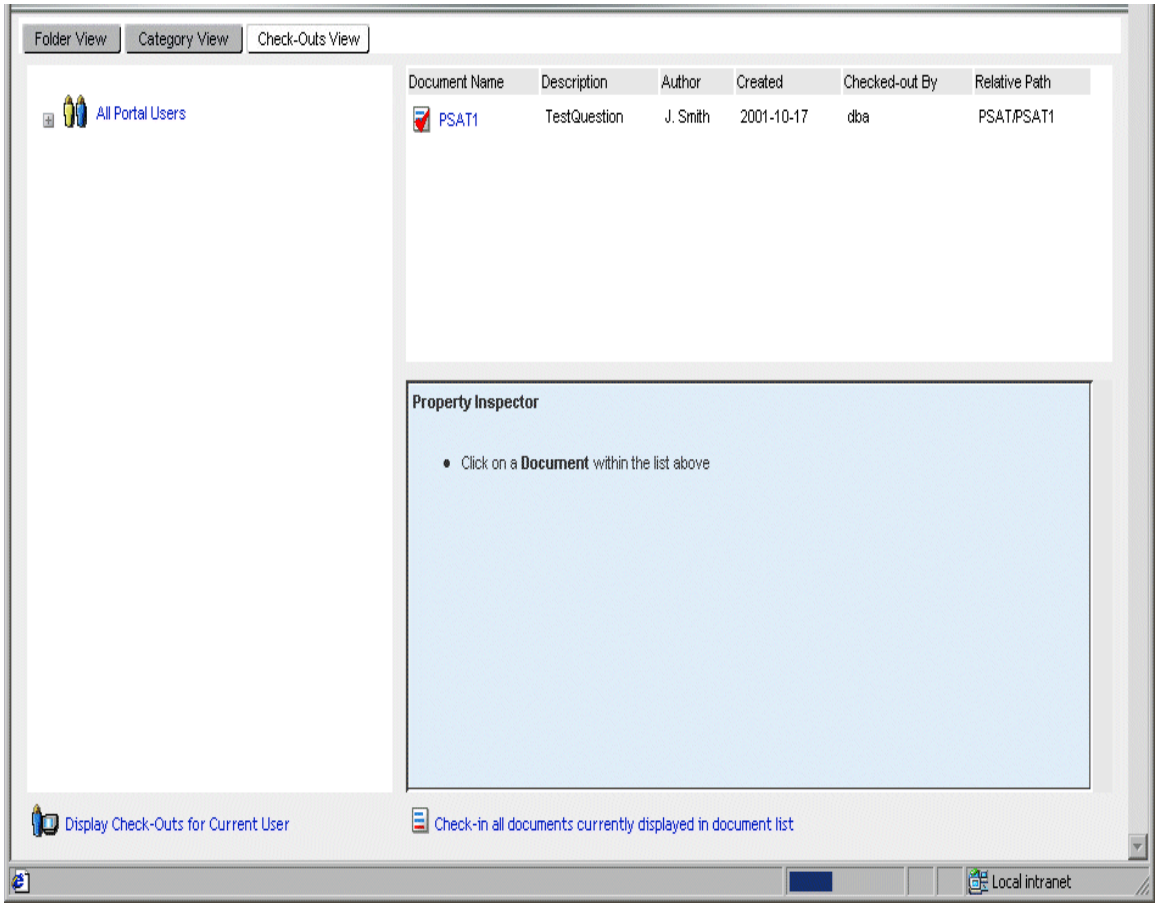
The CMS Administration Console marks checked-out documents for easy identification with a checkmark icon and displays the name of the user who has locked the content. In the following example, the document **PSAT3** has been checked out by user **administrator**:



In this example, the user **dba** now becomes the owner of the document and the only user with authorization to save, delete, edit, and check in the document. If other users try to access PSAT3, they will not see the Save, Delete, Edit, or Check In buttons on the Property Inspector—even if they have WRITE permission for PSAT3—and they will see only the Preview button if they have READ permission for PSAT3.

When a document is checked out, the latest version is locked for editing by the owner. The only way to modify an earlier version of a document is to roll back to that version, as described in "Administering version control" on page 264.

Using the check-outs view The Content tab contains a check-outs view that displays checkouts for either the current user or other users. Here is a sample check-outs view display, with a single file checked out to the current user:



Using the check-outs view, you can:

- View checkouts for the current user or for other exteNd Director users
- View the Property Inspector for the checked-out document by selecting it in the list
- Check in all documents displayed in the list

What happens during checkin

When a document is checked in by its owner, any content modifications are saved as a new version, accessible from the Versions tab in the document's property sheet. Other authorized users are then free to check out the document for editing and will get the most up-to-date version of the content.

Content administrators can implement an automatic checkin feature when they create document types. When this feature is enabled, the CMS Administration Console automatically checks in any document of the specified type after it is edited.

Checkin and checkout procedures

➢ **To check out a document:**

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Navigate to the document of interest and select it to open its Property Inspector.
4. In the Property Inspector, select the **General** tab and click **Check-Out**.

   The CMS Administration Console checks out the latest version of the document, indicating who has locked the content and changing the document icon to the checked-out icon:

➢ **To check in a document:**

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Navigate to the checked-out document of interest and select it to open its Property Inspector.
4. In the Property Inspector, select the **General** tab and click **Check-In**.

   The CMS Administration Console checks in the document, making the most current version of the content available for other users to edit.

➢ **To enable automatic checkin for an existing document type:**

This option is available only to administrators.

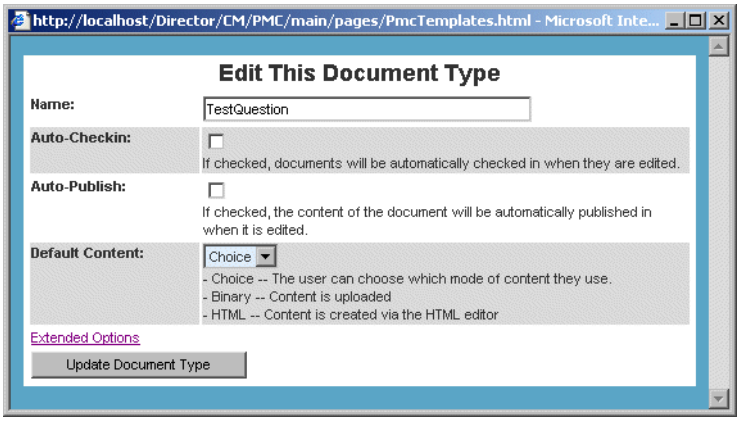
NOTE: You can also set this parameter when you create a new document type, as described in "Creating document types" on page 199.

1 Enter templates mode by clicking the **Templates** button in the toolbar.

A panel appears listing all document types that have been defined.

2 Select the document type for which you want to set automatic check-in, then click **Edit**.

The Edit This Document Type window opens:



3 Check the **Auto-Checkin** check box and click **Update Document Type**.

When you edit a document of this type, the CMS Administration Console automatically checks in your modifications.

Administering version control

The CMS Administration Console provides version control to systematically maintain a history of changes to documents and ensure that the correct content is published.

Administrator tasks The version control system allows administrators with PUBLISH permissions to perform the following tasks:

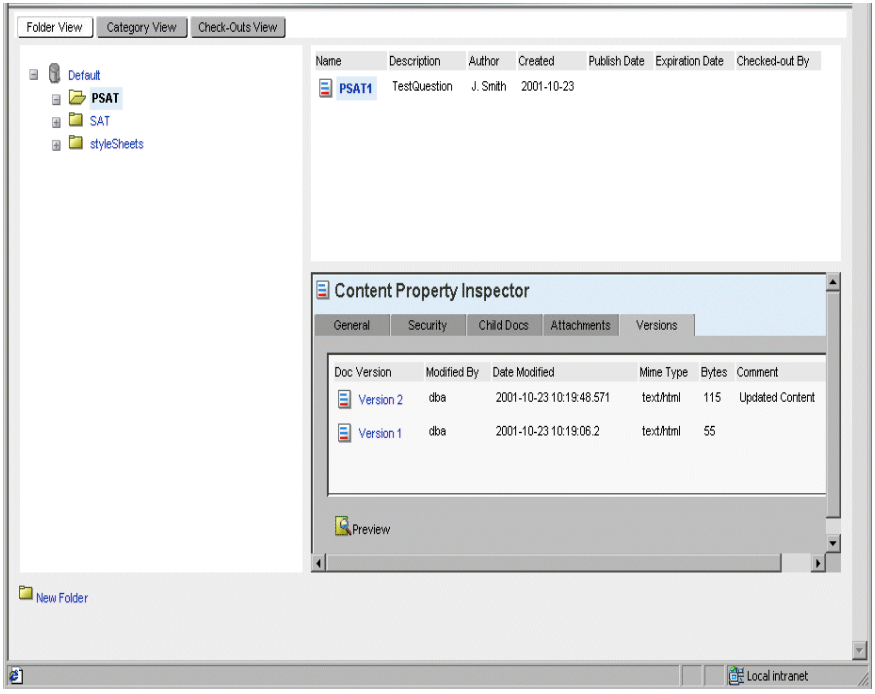
Task	Description
Publish	Approve the designated content version and make the content available for viewing by other users with appropriate permissions. The published version of a document is the content that is returned by the method getContent().

Task	Description
Unpublish	Hide the designated version from public view.
Roll back	Delete all versions of content created after a specified version.

What version you see By default, you receive the latest version of content when you check out and edit a document in the CMS Administration Console. If you want to revert to and modify earlier content, you can roll back to a previous version. Rolling back deletes all later versions of content and sets the target version as the most current.

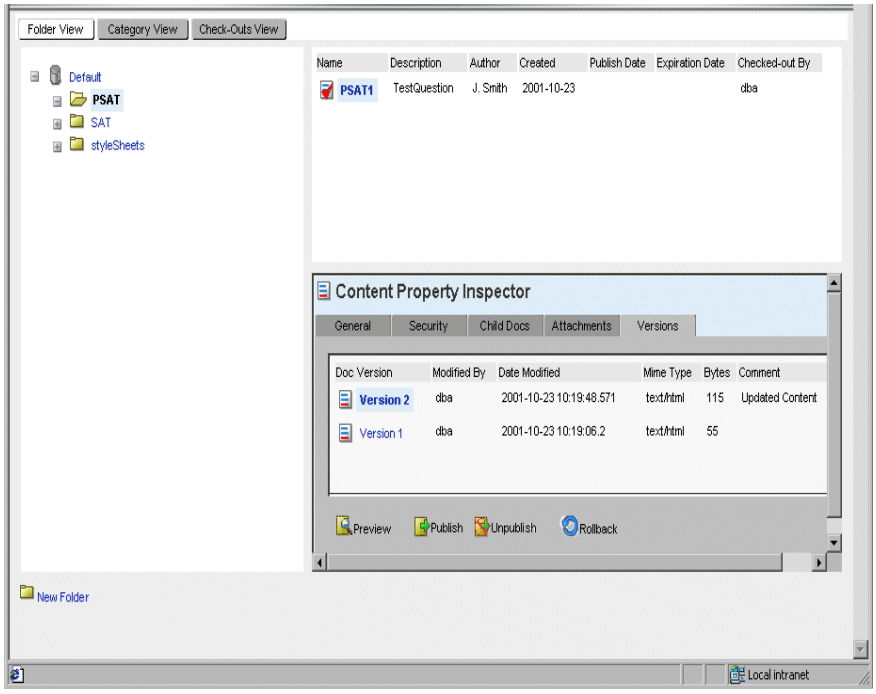
You must check out a document before you can publish, unpublish, or roll back versions of that document. If you have not checked the document out, you can only preview versions of the content.

Any user who opens a document will see a **Versions** tab in the document's Property Inspector. Here is an example of what the Versions panel looks like:



In this example, any user who selects the document PSAT1 can preview its two versions.

Publish features Users with PUBLISH permission can check out PSAT1 and gain the ability to publish, unpublish, and roll back versions, as shown in the refreshed Property Inspector:



Content administrators can also implement an automatic publish feature when they create document types. When this feature is enabled, the CMS Administration Console automatically publishes the content of any document of the specified type if that content is changed.

What's in this section This section explains how to perform the following version control tasks:

- Publish a version
- Unpublish a version
- Roll back to a previous version
- Enable automatic publishing
- Set publish dates

➢ **To publish a version:**

You can publish any version, even if it is not the latest. The CMS Administration Console allows only one version of a document to be published at any given time.

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Navigate to the document of interest and select it to open its Property Inspector.
4. In the Property Inspector, select the **General** tab and click **Check-Out**.

   The CMS Administration Console checks out the latest version of the document.
5. Select the **Versions** tab, then select the document version you want to publish.
6. Click **Publish**.

   The CMS Administration Console publishes the version you selected, marking it with the published-version icon:

   
7. Return to the **General** tab and click **Check-In**.

   The published version cannot be edited, even when the document is checked out.

➢ **To unpublish a version:**

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Navigate to the document of interest and select it to open its Property Inspector.
4. In the Property Inspector, select the **General** tab and click **Check-Out**.

   The CMS Administration Console checks out the latest version of the document.
5. Select the **Versions** tab, then select the published version you want to unpublish.

   Published versions appear with this icon:

   
6. Click **Unpublish**.

   The CMS Administration Console unpublishes the version you selected, marking it with the default document icon:

   
7. Return to the **General** tab and click **Check-In**.

➢ **To roll back to a previous version:**

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Navigate to the document of interest and select it to open its Property Inspector.
4. In the Property Inspector, select the **General** tab and click **Check-Out**.

   The CMS Administration Console checks out the latest version of the document.
5. Select the **Versions** tab, select the version you want to roll back to, then click **Rollback**.
6. When a confirmation window opens, click **OK**.

   The CMS Administration Console deletes all versions created after the selected version—even if one of these later versions was already published. The selected version becomes the latest version.
7. Return to the **General** tab and click **Check-In**.

➢ **To enable automatic publish:**

Only users with administrative permissions can implement this feature. Enabling automatic publish produces the following effects:

- Whenever you edit the **dynamic content** of a document, the CMS Administration Console automatically publishes a new version of the document.
- If you edit only the **metadata and field portions** of a document, the CMS Administration Console automatically updates and publishes the latest version of the document.

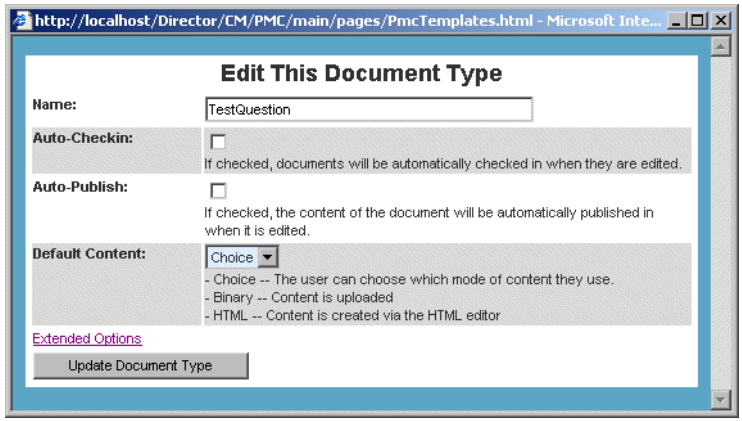
➢ **To enable automatic publish:**

1. Enter templates mode by clicking the **Templates** button in the toolbar.

   A panel appears listing the document types that have been defined.

2. Select the document type for which you want to set automatic checkin, and click **Edit**.

   The Edit This Document Type window opens:



3. Check the **Auto-Publish** check box and click **Update Document Type**.

   When you edit and save a document of this type, the CMS Administration Console automatically publishes your modifications as a new version of the content. This latest version becomes the published version, regardless of whether an earlier version was already published or no earlier versions were published.

NOTE: You can also enable automatic publish when you create a new document type, as described in "Creating document types" on page 199.

➢ **To set publish dates:**

The CMS Administration Console does not automatically set publish dates, although content administrators with WRITE permission can set publish dates manually anytime to mark documents for publication. After that, developers can write scheduled business objects that publish documents based on these dates.

1. Enter Content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your existing folders appear in the content tree view. You may need to expand some of these containers to see the complete view.

3. Navigate to the document of interest and select it to open its Property Inspector.
4. In the Property Inspector, select the **General** tab and click **Check-Out**.

   The CMS Administration Console checks out the latest version of the document.

5 In the **Publish Date** field, enter a publish date of the form:

```
YYYY-MM-DD HH:MM:SS
```

6 Click **Save** to record the date.

18 Searching Content

This chapter describes how to use the Autonomy search engine to search content in the CMS Administration Console. It has these sections:

- Setting up the CMS Administration Console search facility
- Using the search facility in the CMS Administration Console
- Search options

Setting up the CMS Administration Console search facility

The search facility of the CMS Administration Console uses the Autonomy search engine (Dynamic Reasoning Engine, or DRE). The Autonomy DRE uses conceptual pattern matching, which is a more sophisticated form of searching than keyword-based full-text searching.

Before you can use the search facility in the CMS Administration Console, you must:

- Configure the Autonomy DRE to interface properly with your server.

  For instructions on configuring Autonomy for use with your server and the CM subsystem, see the section on configuring your environment in the *Content Search Guide*.

- Configure the CM subsystem to link to the Search service for your exteNd Director EAR project.

  You can make search configuration settings when you create your project. After you have created a project, you can change search configuration settings for the CM subsystem in the exteNd Director EAR configuration tool.

  NOTE: For changes in content to be immediately available to the CMS Administration Console's search facility, you must set Synchronization Mode to **immediate** and select which document operations you want to trigger immediate synchronization (for example, checkin and publish). In **batch** mode, changes are propagated to the DRE by the synch task.

- If you have made configuration changes in an existing project, redeploy your project.

After you have configured you environment for the Autonomy DRE and configured the search options for your project, you can use the search facility in the CMS Administration Console.

Using the search facility in the CMS Administration Console

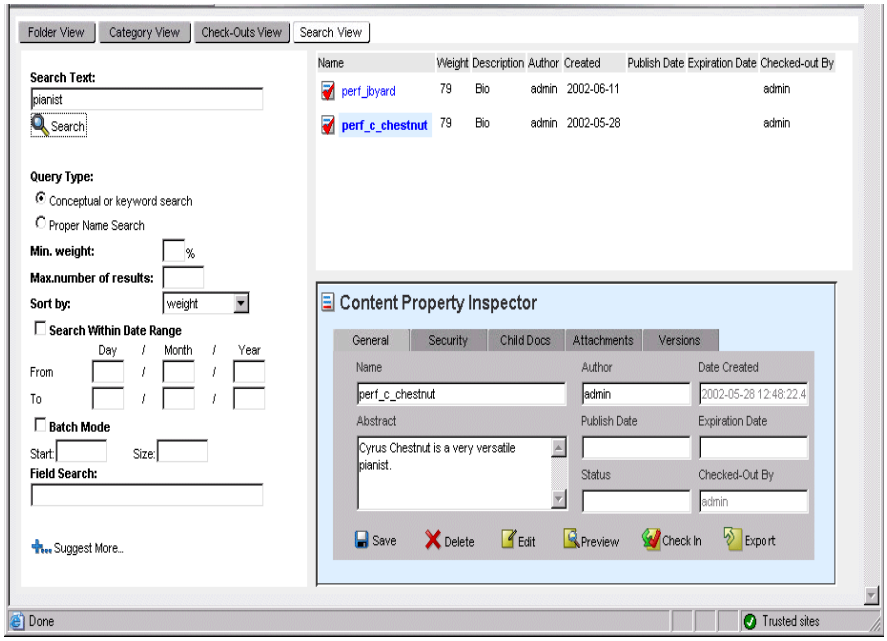
➢ **To perform a search in the CMS Administration Console:**

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Search View** tab.
3. In the Search Pane:
   - **3a** Enter the word or phrase you want to search for in the **Search Text** box.
   - **3b** (Optional) Set any other search options you want to use to refine your search. See "Search options" on page 274.
   - **3c** Click the **Search** button:

The search view In the search view:

- The Search Pane replaces the content tree.
- Documents found by the search are listed in the content list. If you click a document to select it from the list, the Content Property Inspector appears, as shown in Step 2 above.
- In the content list, there is a **Weight** column between the Name and Description columns. The numbers in the Weight column indicate the relevance of each found document to the search criteria, expressed as a percentage:

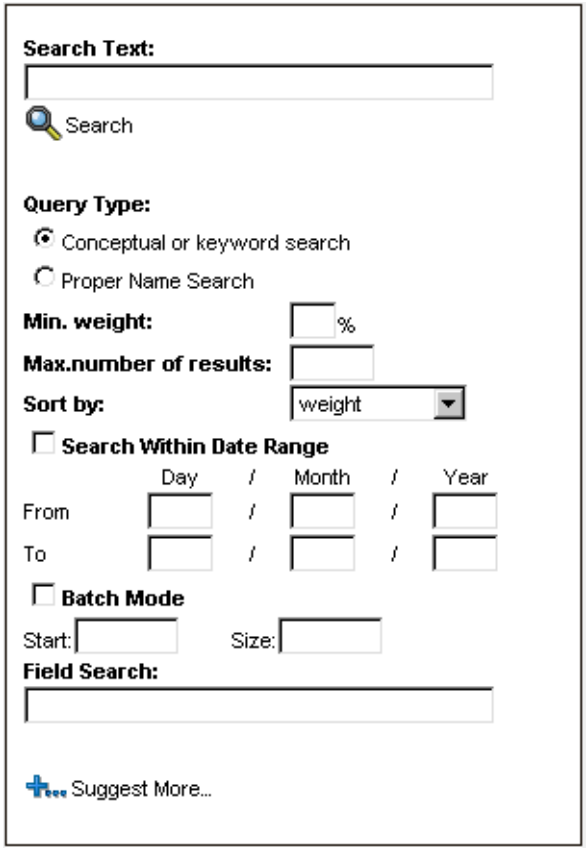


If you are not getting the results you expect If the search facility is not finding documents you expect it to find:

- Make sure you have created or imported content into your repository.
- Review the synchronization settings for your project. You can do this in the exteNd Director EAR Configuration tool.
- If you are using immediate synchronization, make sure you have made your documents available to the search engine by performing one or more of the operations you chose when you configured immediate synchronization on each of the documents.

Search options

In the Search Pane you can set a number of options to refine your search:



The following table explains how to use the search options:

Option	How to use the option
Search Text	Enter the word or phrase you want to search for.

Option	How to use the option
Query Type	Select the type of search you want to perform: ◆ Conceptual or keyword search (the default)—When this type is selected, the DRE uses conceptual pattern matching by default. If you use semicolon notation (for example: `silk;+worm;`) the search engine performs a keyword search based on the number of occurrences of the terms, rather than on their conceptual relevance. ◆ Proper Name Search—When this type is selected, the search engine treats the search text as a proper name, and performs a conceptual search accordingly.
Min. weight	Enter the minimum weight for a document to be displayed in the content list. The *weight* of a found document is a measure of its relevance to the search text. The search engine assigns a percentage value to each document, with 100% representing the greatest possible relevance.
Max. number of results	Enter a number that specifies the greatest number of documents you want to be displayed in the content list.
Sort by	Select a sort order from the dropdown list. The available choices are: ◆ weight (the default) ◆ date ◆ weight and date
Search Within Date Range	Select this check box if you want to restrict the search to documents created within a specified time period. For both the From and To dates, enter the day, month, and year in the corresponding text boxes.

Option	How to use the option
Batch Mode	Check this check box if you want a subset of the found documents to appear in the content list. When using batch mode, it is helpful to think of the full set of found documents as an array, ordered according to the sort order you indicate in the Sort by box. The documents that are displayed are selected from the full set of found documents, based on the numeric values you enter in the Start and Size boxes: ◆ Start—Specifies the position of the first document (from the full set of found documents) to be displayed in the content list. Like array elements, the order of the documents in the full set of found documents begins with 0. ◆ Size—Specifies the total number of documents you want to be displayed in the content list, beginning with the document specified by the Start value. **Example** Say you perform a search without using batch mode that returns six documents. Then you repeat the search in batch mode, indicating a Start value of 1 and a Size value of 3. The search now returns the second, third, and fourth documents from the original set of found documents, based on the order in which they initially appeared in the content list.

Option	How to use the option
Field Search	Enter a field search expression. The syntax of a field search expression is: *fieldname1=value1 operator fieldname2=value2 ...* where: ◆ **fieldname** is the name of an extension metadata field you have created, or one of these standard metadata fields: ◆ AUTHOR ◆ CONTENTSIZE ◆ CREATED ◆ DOCABSTRACT ◆ DOCID ◆ DOCNAME ◆ DOCTYPEID ◆ DOCTYPENAME ◆ EXPIRATIONDATE ◆ FOLDERID ◆ LOCKEDBY ◆ MIMETYPE ◆ PARENTDOCID ◆ PUBLISHDATE ◆ PUBLISHSTATUS ◆ STATUS ◆ SUBTITLE ◆ TITLE ◆ UPDATETIME ◆ **value** is the field value you are searching for ◆ **operator** is either: ◆ AND or ◆ OR **Example** If you want to limit your search to all HTML documents written by user admin, the field search expression you would use is: `author=admin AND mime-type=text/html`

Option	How to use the option
Suggest More	If you want to find documents related to a document that was found by previous search, select that document in the content list and click the Suggest More button. The list of documents found by the previous search is replaced by a list consisting of the selected document and any related documents.

19 Managing Content Security

This chapter describes how to secure access to content using the CMS Administration Console. It has these sections:

- About content security
- Flow of operations
- Permissions for content access
- User permissions required for CM operations
- Cascading security
- Setting security on CM elements

For background information, see Chapter 4, “Securing Content”.

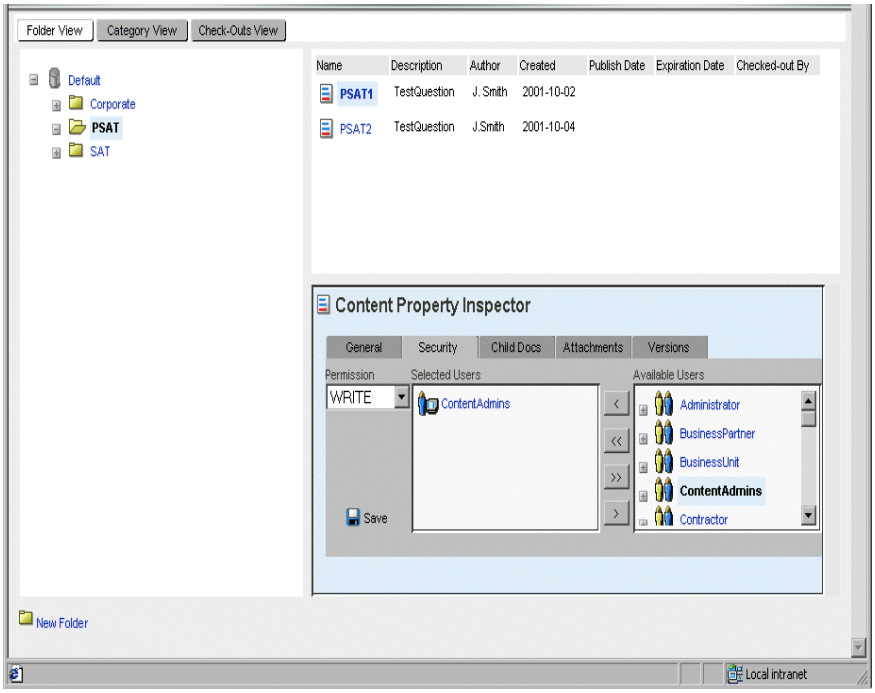
About content security

The CMS Administration Console allows administrators—and other users with PROTECT permission—to control access to CM elements. Administrators can assign users and groups various levels of access permission on an element-by-element basis to the following types of content:

- Document
- Folder
- Taxonomy
- Category

When users with PROTECT permission open one of these CM elements in the CMS Administration Console, they will see a **Security** tab in the Property Inspector. The Security tab displays controls for assigning levels of access to the selected CM element.

For example, here is what the Security tab looks like after assigning the ContentAdmins Group WRITE access to the document PSAT1:

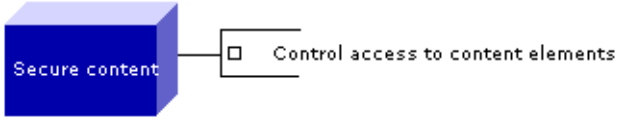


The CMS Administration Console provides security-sensitive controls as part of its user interface. It gives you only those CM capabilities that are permitted by the security privileges assigned to you for each CM element.

For example, if you have WRITE permission for all documents, you can check out and edit any document in the CMS Administration Console. If you do not have WRITE permission for documents in a confidential folder, you will never see **Edit** and **Check-Out** controls in the Property Inspectors of documents residing in that folder.

Flow of operations

Here is the basic task for securing content in the CMS Administration Console:



This chapter explains how to manage security in the CMS Administration Console and includes the following topics:

- Permissions for content access
- User permissions required for CM operations
- Cascading security
- Setting security on CM elements

Permissions for content access

Administrators with PROTECT permission can assign users various levels of content access based on their roles in the organization.

The CMS Administration Console allows authorized users to assign the following access permissions:

Permission	Allows you to
READ	View any data and/or metadata associated with the designated CM element—for example, preview a document or view the metadata associated with a folder
WRITE	Create, modify, and save the designated CM element
PROTECT	Set security on a designated CM element
LIST	View lists of documents in a folder or category **NOTE:** This permission applies to folders or categories only, not to documents.
PUBLISH	Publish a document **NOTE:** This permission applies to documents only, not to folders or categories.

While each of these access permissions is assigned to CM elements individually (as described in "Setting security on CM elements" on page 284), it is not necessary to explicitly set access permissions on each element. A CM element can inherit access permissions from its parent element.

For more information on setting CM element permissions through inheritance, see "Cascading security" on page 283.

User permissions required for CM operations

The following table describes which permissions are required for performing specific CM operations in the CMS Administration Console:

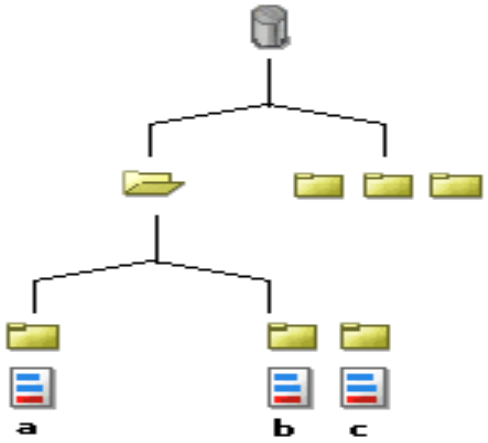
Element	Operation	Permission
Document	View content or metadata	READ
	Modify content or metadata	WRITE
	Publish	PUBLISH
	Set security	PROTECT
Folder	View metadata	READ
	Modify folder metadata Add subfolder Add document	WRITE
	List contents	LIST
	Set security	PROTECT
Category	View metadata	READ
	Modify category metadata Add subcategory Add document	WRITE
	List contents	LIST
	Set security	PROTECT
Field	View metadata	READ
	Modify metadata	WRITE
	Set security	PROTECT
Document type	View metadata	READ
	Modify metadata	WRITE
	Set security	PROTECT
	List fields that belong to the document type	LIST
Layout style	View metadata	READ
	Modify metadata	WRITE
	Set security	PROTECT

For information on giving users and groups levels of access to individual CM elements, see "Setting security on CM elements" on page 284.

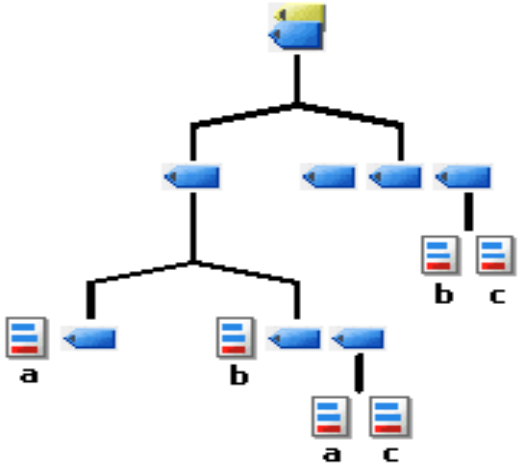
Cascading security

Generally, security settings cascade from parent to child in the hierarchical relationships of CM elements. The following content hierarchies exist in the CMS Administration Console:

- Physical hierarchy of root folders, folders, and documents:



- Logical hierarchy of taxonomies, categories, and documents:



Inherited security When a new child is created in either hierarchy, it inherits the parent's security settings. Child elements can also inherit changes to a parent's access permissions, but you must explicitly enable this behavior, as described in "Setting security on CM elements" on page 284.

Setting security on CM elements

Users with PROTECT permission can set security on the following CM elements:

- Documents
- Folders
- Categories
- Taxonomies

➢ **To set security on documents and folders:**

1. Enter content mode by clicking the **Content** button in the toolbar.
2. Select the **Folder View** tab.

   Your folders appear in the content tree view. You may need to expand some of these containers to see the complete view.
3. Navigate to the folder or document of interest and select it to open its Property Inspector.
4. Select the **Security** tab.
5. Select a permission from the dropdown list.
6. Assign this permission to the appropriate users and groups by following these steps:

To	Do this
Assign individual users and groups	1 Select users or groups one at a time from Available Users. 2 Click the single-arrow button to move each selection to Selected Users. **NOTE:** You cannot multiselect users and groups from Available Users.
Assign all users and groups	Click the double-arrow button. **NOTE:** All groups move from Available Users to Selected Users.

7 To allow existing children of the selected folder to inherit the new security setting, check **Apply Security To Existing Children**.

IMPORTANT: This option is available only to administrators.

8 Click **Save**.

➢ **To set security on categories and taxonomies:**

1 Enter content mode by clicking the **Content** button in the toolbar.

2 Select the **Category View** tab.

Your categories and taxonomies appear in the content tree view. You may need to expand some of these containers to see the complete view.

3 Navigate to the category or taxonomy of interest and select it to open its Property Inspector.

4 Select the **Security** tab.

5 Select a permission from the dropdown list.

6 Assign this permission to the appropriate users and groups by following these steps:

To	Do this
Assign individual users and groups	1 Select users or groups one at a time from Available Users. 2 Click the single-arrow button to move each selection to Selected Users. **NOTE:** You cannot multiselect users and groups from Available Users.
Assign all users and groups	Click the double-arrow button.

7 To allow existing children of the selected folder to inherit the new security setting, check **Apply Security To Existing Children**.

IMPORTANT: This option is available only to administrators.

8 Click **Save**.

20 Importing and Exporting Content

This section describes how to import and export content using the CMS Administration Console:

- About the import and export facilities
- Summary of CMS Administration Console import and export behavior
- Exporting content
- Importing content
- Structure of the data import or export archive
- Best practices and prerequisites

 For background information about how the functions work and how to customize the import and export functions, see Chapter 7, “Importing and Exporting Content”.

About the import and export facilities

The CMS Administration Console allows you to **export** CM data from your repository, beginning from any point in the Content Tree. You can also export the entire contents of a CM system from the toolbar.

Similarly, the CMS Administration Console allows you to **import** CM data at any point in the Content Tree, or the entire contents of a CM system from the toolbar.

Uses for the import and export facilities include:

- Moving or copying folders, categories, and documents within a repository
- Moving CM data between different stages of development
- Integrating with third-party vendors

- Backing up and restoring CM data
- Debugging and data analysis

Import and export of CM infrastructure It is also possible to export and import all or part of the supporting infrastructure of your CM subsystem, such as fields or document types.

Import and export of archives When you export CM data from the CMS Administration Console, it is stored in a ZIP file that serves as a structured export archive. When you import CM data using the CMS Administration Console, it must be imported from a ZIP file that follows the same structure as the export archive. When you import CM data that has been exported from a CM repository, you import directly from the export archive.

Summary of CMS Administration Console import and export behavior

Here is what happens when you export or import CM data, depending on the starting point for the operation:

Starting point	Export: what goes into the ZIP file	Import: where the contents of the ZIP file are placed
Toolbar	The entire contents of the CM subsystem including: ◆ The Content Admin element ◆ Taxonomies ◆ Categories ◆ Display styles ◆ Document types ◆ Fields ◆ Folders ◆ Documents ◆ Document versions **OR** A subset of the CM subsystem, as specified by a document export descriptor (DED)	The Default folder

Starting point	Export: what goes into the ZIP file	Import: where the contents of the ZIP file are placed
Repository Property Inspector	All folders, documents, document versions, fields, and document types contained in the repository	The Default folder
Folder Property Inspector	The selected folder and all its contents, including: ◆ Documents and associated versions, fields, and document types ◆ Subfolders of the selected folder, and their contents	The selected folder
Content Property Inspector	All versions of the selected document, plus any document type and fields associated with it	Not applicable

 For more information on what goes into the export archive and how the archive is structured, see "Structure of the data import or export archive" on page 297.

Exporting content

This section explains how to export CM data from the toolbar and the Property Inspectors.

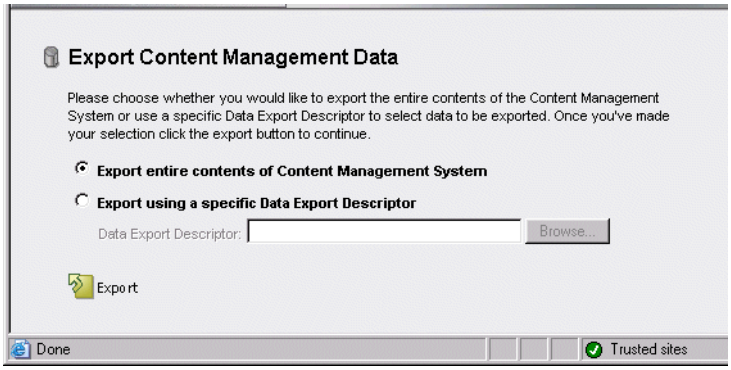
NOTE: Before you export data, be sure to review the section "Best practices and prerequisites" on page 298.

Exporting from the toolbar

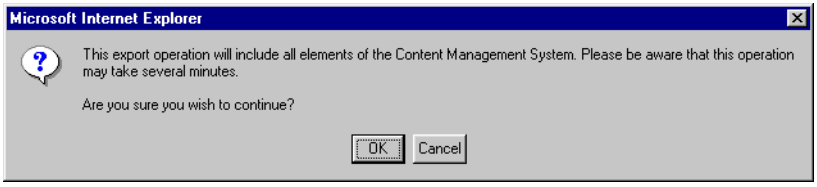
The **Export** button on the toolbar allows you to export the entire contents of your CM subsystem, or to perform a customized export using a descriptor file called the *data export descriptor* (DED).

➢ **To export content from the toolbar:**

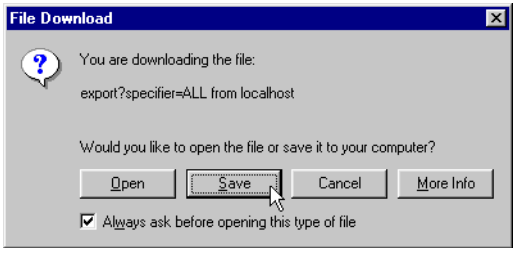
1. Click the **Export** button on the CMS Administration Console toolbar.
   The Export Pane displays:



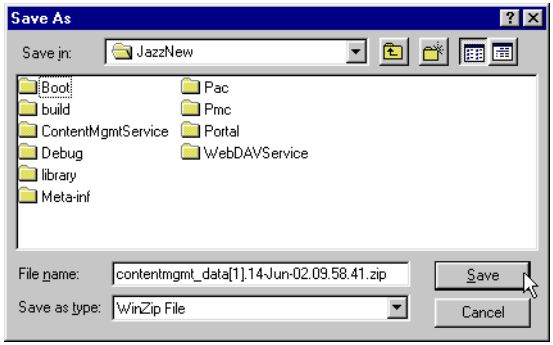
2. Choose **Export using a specific Data Export Descriptor**.
3. Click the **Browse** button and navigate to the DED file you want to use for this export.
4. Click the **Export** button.
5. Click **OK** in the question box that appears:



6. In the File Download dialog, click **Save**:



7 In the Save As dialog, navigate to the folder where you want to store the export archive, give the archive file a unique name, and click **Save**.

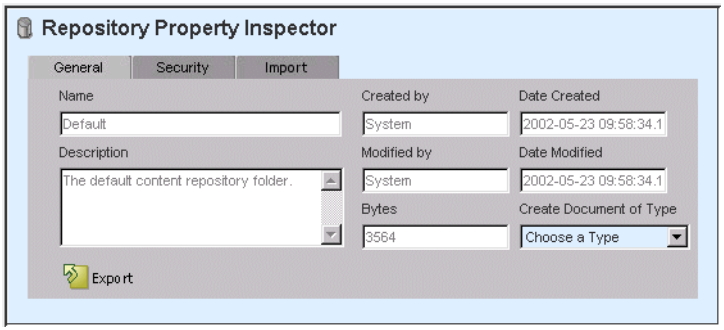


For more information about the archive, see "Structure of the data import or export archive" on page 297.

Exporting from a Property Inspector

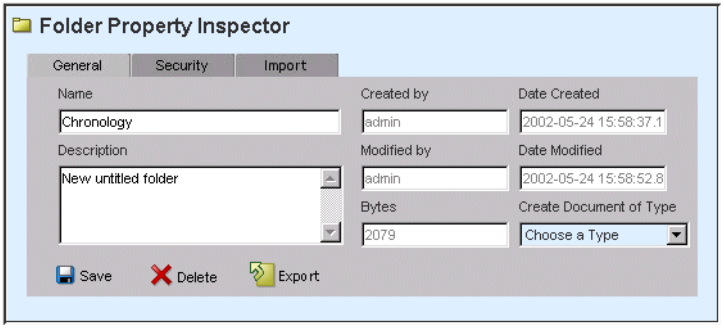
➢ **To export content from a Property Inspector:**

1. Enter Content mode by clicking the **Content** button on the CMS Administration Console toolbar.
2. Select the starting point for the export by doing **one** of the following:
   - In the Content Tree, click the **Default** folder. The Repository Property Inspector displays:



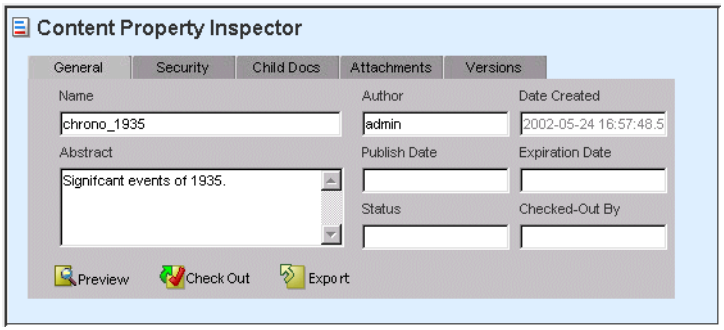
OR

   - In the Content Tree, click any folder other than the Default folder. The Folder Property Inspector displays:



OR

- In the content list, click a document. The Content Property Inspector displays:



3 Click the **Export** button in the Property Inspector.

4 Follow Step 5, Step 6, and Step 7 in "To export content from the toolbar:" on page 290 to name and save your export archive.

📖 For a description of the contents of the export archive file, see "Structure of the data import or export archive" on page 297.

Customizing exports

You can configure and customize the export process by editing the DED.

📖 For more information, see "Customizing imports and exports" on page 113.

Importing content

This section describes the import process and explains how to import data into your CM subsystem from the toolbar and the Property Inspectors.

NOTE: Before you import data, be sure to review the section "Best practices and prerequisites" on page 298.

Data not previously exported If you want to import data that **was not** previously exported from a CM repository, you can do this manually by assembling an import ZIP file, or programmatically using the CM API. For more information, see "Customizing imports and exports" on page 113.

Data previously exported If you are importing data that **was** previously exported from a CM repository—for example, as part of a moving or copying process—you import directly from the export archive so that the archive will automatically follow the required structure.

Configuring the import process

Unlike with exporting content (when you can configure the process only from the toolbar), when you are importing content you can configure the process regardless of the starting point. You do this by adding a *data import descriptor* (DID) to the import archive file or editing the existing file before performing the import.

NOTE: When you are importing previously exported CM data, the import archive will always contain a DID (called **contentmgmt_did.xml**) in the **contentmgmt-inf** folder.

For more information about the DID, see "Customizing the data import descriptor (DID)" on page 114.

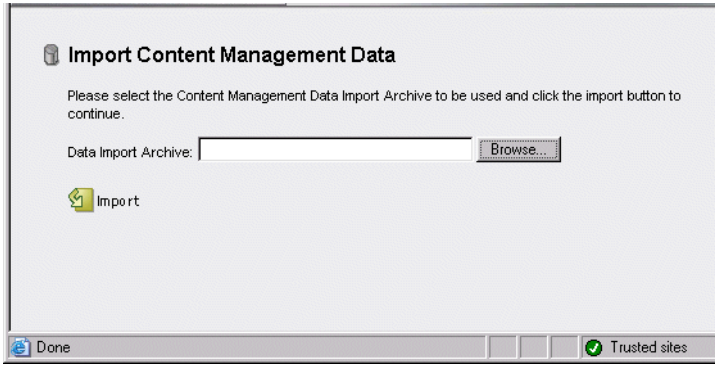
Importing from the toolbar

The **Import** button on the toolbar allows you to import CM data from an import archive into the Default folder of a repository.

➢ **To import content from the toolbar:**

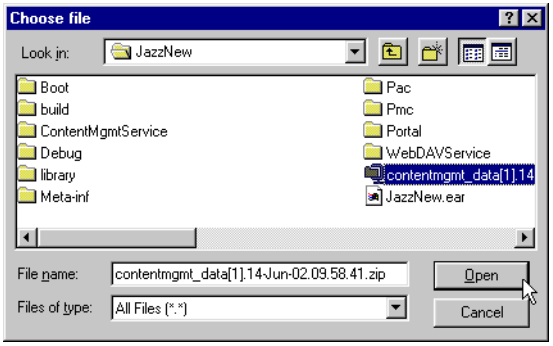
1. Click the **Import** button on the CMS Administration Console toolbar.

   The Import Pane displays:



2. Click the **Browse** button.

3 In the Choose File dialog, browse to the import archive you want to use and click **Open**:



4 In the Import Pane, click **Import**.

Importing from a Property Inspector

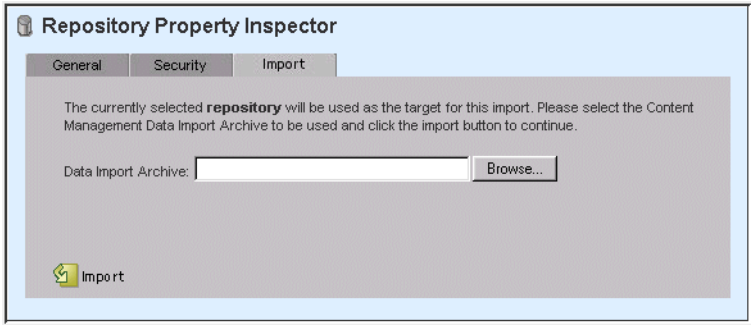
You can import from the Repository Property Inspector and the Folder Property Inspector (but not from the Content Property Inspector).

➢ **To import content from a Property Inspector:**

1 Enter content mode by clicking the **Content** button on the CMS Administration Console toolbar.

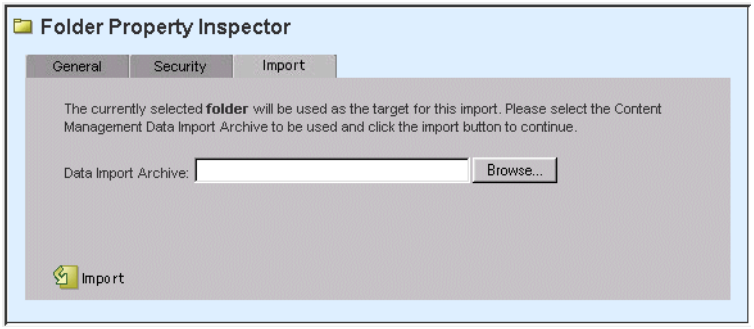
2 Select the starting point for the import by doing **either** of the following:

- In the Content Tree, click the **Default** folder. The Repository Property Inspector displays:

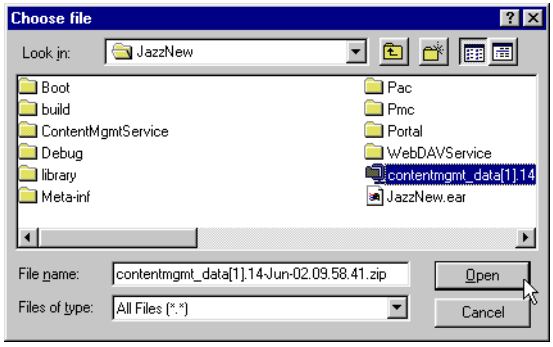


OR

- In the Content Tree, click any folder other than the Default folder. The Folder Property Inspector displays:



3. In the Property Inspector, click the **Import** tab.
4. Click **Browse**.
5. In the Choose File dialog, browse to the import archive you want to use and click **Open**:



6. In the Import pane, click **Import**.

Structure of the data import or export archive

The following table shows the internal folder structure of a data import or export archive file and explains what each folder contains:

Folder name	Contains	Included when exporting from:	
		Toolbar (entire system)	Repository, Folder, or Content property inspector
contentmgmt-inf	contentmgmt_did.xml (the DID)	●	●
admin_metadata	ContentAdmin.xml (the Content Admin element)	●	
categories_metadata	XML descriptor files for each taxonomy and category, organized according to the structure of the taxonomy(ies)	●	
styles_metadata	An XML descriptor for each style, registering its name and listing the document type it is associated with	●	
fields_metadata	An XML descriptor for each field, registering the field name and the data type of its value	●	●
fields_data	The application-specific data associated with each extension metadata field; for fields created with the CMS Administration Console, this consists of an XML descriptor for each field listing its properties, including its control type and (if applicable) its possible values	●	●
doctypes_metadata	An XML descriptor for each document type, listing the fields associated with it	●	●
doctypes_data	The application-specific data associated with each document type; for document types created with the CMS Administration Console, this consists of an XML descriptor for each document type describing its properties	●	●
folders_metadata	An XML descriptor for each folder, registering the folder and listing its parent folder, if any	●	●

Folder name	Contains	Included when exporting from: Toolbar (entire system)	Included when exporting from: Repository, Folder, or Content property inspector
docs_metadata	An XML descriptor for each document containing the names and values of the fields associated with the document, organized according to the folder structure	●	●
docs_content	Files containing the published content of each exported document, organized according to the folder structure	●	●
docs_content_versions	Files containing the content of each version of exported document, organized according to the folder structure	●	●

Best practices and prerequisites

This section provides some notes on best practices for importing and exporting CM data.

Planning for large-scale import/export operations

If you are planning to export or import a very large amount of CM data, it is important to keep the memory capacity of your machines in mind as you plan your operation.

During an import or export operation, all objects representing elements of the repository must be present in memory at the same time. That means the amount of available memory imposes a practical limit on the size of a repository you can process in a single operation.

The best way to approach a large-scale operation is to export or import your source repository in logical chunks. For example, you might export all your document types in one operation, your fields in another operation, and so on, ending with exporting or importing your document content in manageable chunks according to the folder structure of your repository.

Security considerations

This section applies primarily to importing CM data that has been exported from another repository.

Permissions to establish in the import target The user who performs the export from the source repository must exist and must have the SearchAdmin WRITE permission in the target repository.

Users to create in the import target You need to make sure that if any documents were checked out at the time of export, the users to whom they are checked out have been created in the repository into which you are importing.

If these users do not exist in the import repository, the import will fail.

21 Administering Automated Tasks

Tasks mode in the CMS Administration Console allows you to view, start, and stop automated CM tasks from the CMS Administration Console. This chapter includes these topics:

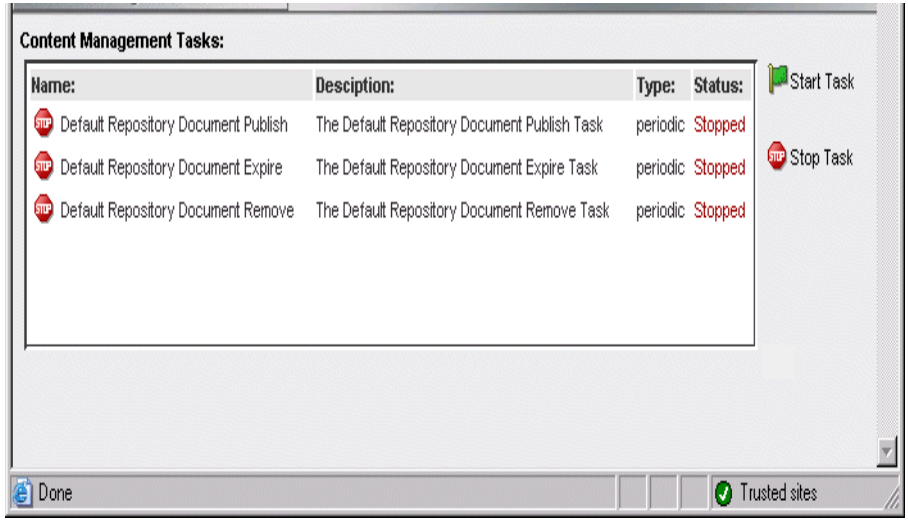
- The task display
- Starting and stopping tasks

Several tasks are installed with the CM subsystem. You can modify these installed tasks and/or create new, custom tasks to meet the specific needs of your application.

📖 For more information, see Chapter 5, “Managing Tasks”.

The task display

You enter tasks mode by clicking the **Tasks** button on the CMS Administration Console toolbar. The task display appears, as in this example:



This display provides the following information about the tasks defined on your server:

Task property	Details	Example
Name and description	As defined in the task object.	—
Type	The task type, from a scheduling point of view. Possible types are: ◆ **Periodic**: a task that is scheduled to run multiple times at regularly scheduled intervals. For example, a periodic task could be a repository backup utility that runs every 24 hours (86,400,000 milliseconds). ◆ **Scheduled**: a task that is scheduled to run at one or more fixed points in time.	For example, a scheduled task could be a content publishing task that is scheduled to run at three publication deadlines, such as: ◆ Monday, June 24, 2002 at 9 a.m. ◆ Wednesday, June 26, 2002 at 5 p.m. ◆ Friday, June 28, 2002 at midnight

Task property	Details	Example
Status	The execution status of the task. Possible values are: ◆ **Stopped**: Task is not yet running or has been halted. ◆ **Started**: Task is currently running.	—

Starting and stopping tasks

Tasks are not persistent across application server sessions. Each time you restart your server, you must restart each of your tasks.

➢ **To start or stop a task:**

1. Enter tasks mode by clicking the **Tasks** button in the toolbar.
2. Click anywhere in a task description to select it.
3. Click the **Start Task** button to start the task or the **Stop Task** button to stop the task.

IV Applications

Describes how to use the Content Query and RSS portlet application

- Chapter 22, “Content Query Application”

22 Content Query Application

This chapter describes how to use the Content Query action and related artifacts to query the Content Management subsystem. It has these sections:

- About Content Query
- Using the Content Query action

NOTE: To use this application your project must include the Content Management and the Rule subsystems.

About Content Query

The Content Query action (CQA) allows you to query published documents in the Content Management (CM) subsystem. You can query by folder, category, document type, or by specific document. Searches can be designated as either inclusive or exclusive. The results of the query are captured in XML and processed as a query in the CM subsystem.

Content Query consists of a portlet and sample rules that use the installed Content Query action. The application artifacts are provided in your exteNd Director directory at:

```
Portal/WEB-INF/lib/cqa-portlets.jar
```

Application contents The CQA-Portlets JAR includes:

CQA Artifact	Description
ContentListPortlet.class	Portlet that displays the results of a query using the Content Query action.

CQA Artifact	Description
ContentList.xml	Sample rule that executes a general document query against the CM subsystem. 📖 See Using the Content Query action.
MyDocuments.xml	Sample rule that executes a query for documents created or modified by the logged-in user
NewDocuments.xml	Sample rule that executes the SetDateonWhiteboard action and executes a query for documents created or modified on the current date

Using the Content Query action

The Content Query action provides a custom user interface in the Rule Editor for specifying the folders, categories, document types, and documents to include (or exclude) in the query results. It also provides an interface for selecting the properties (content fields) that should be displayed in the query output and for specifying sort rules. The Content Query action also includes a query builder to allow you to specify selection criteria.

➢ **To edit and run a query:**

1. If you have not yet created content, you need to add some content using the Director Administration console (DAC) or WebDAV.
2. Start your server and open the ContentList rule in exteNd Director.

   📖 For more information, see the section on using the rule and macro editors in the *Rules Guide*.

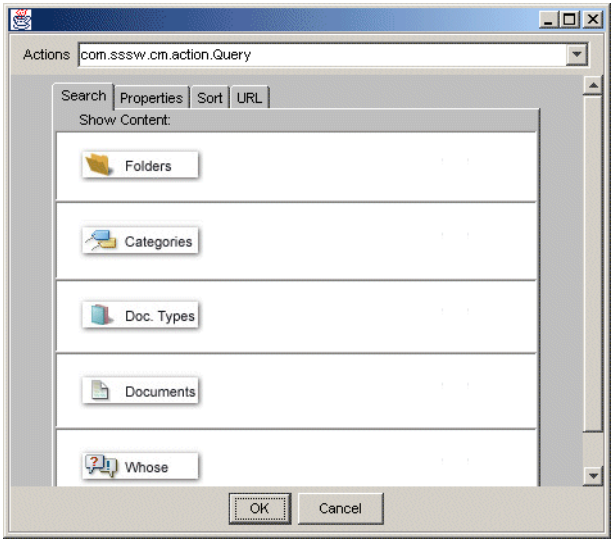
   NOTE: You can also create a new rule and add the Query action. If you are creating your own rule, skip the next step.
3. Select the **Edit query against the content management system** action, then right-click and select **Edit** from the popup menu.

   A popup asks you to specify the URL to your project's ContentMgmtService folder.

4 Specify the correct URL—for example:

```
http://localhost/MyDirectorProj/ContentMgmtService/
```

The Content Query Property Inspector displays:



5 On the **Search** tab, specify which documents you want to include (or exclude) in your query:

To select one or more	Click the
Folders to be included or excluded	**Folders** button
Categories to be included or excluded	**Categories** button
Document types to be included or excluded	**Doc Types** button
Documents to be included or excluded	**Documents** button

Each property panel allows you to specify an URL to a whiteboard key for the documents:

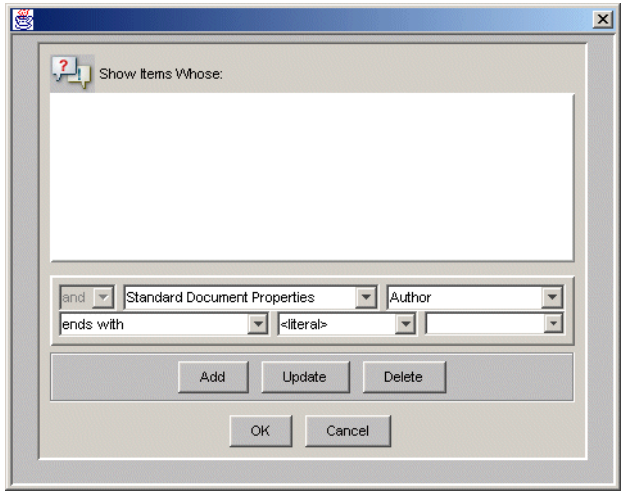
You can either enter the value or specify a whiteboard key that holds the value you want. Use this format:

```
!valueOf.keyname
```

You can also specify a key that holds the name of another key. To get a value from another key, specify !valueOf.*anotherkey*.

 For more information about the !valueOf construct, see the section on using whiteboard values in the *Rules Guide*.

6 To build a query condition, click the **Whose** button:



The Whose query builder lets you specify selection criteria for individual CM properties. To build a query condition:

Step	Action
1	If you've already added one or more conditions to the query, select a logical operator (**and** or **or**).
2	Select **Standard Document Properties**.
3	Select a property (such as **Author**).
4	Select an operator (such as **ends with**).
5	Select **<literal>**. **TIP:** Only literal strings or whiteboard keys are supported at this time.
6	Enter a value that will be used for the expression. You can either enter the literal value or a whiteboard key that holds a value. Use this format: `!valueOf.`*`keyname`* You can also specify a key that holds the name of another key. To get a value from another key, specify !valueOf.*anotherkey*. For more information about the !valueOf construct, see the chapter on using whiteboard values in the *Rules Guide*.
7	Click **Add** to add the condition.

The query specifications you provide on the Search tab are ANDed together. That means that to be included in the result set for the query, a document must satisfy all criteria specified on the Search tab.

7 On the **Properties** tab, select the document properties that you want to appear in the query output. You can select one or more properties from the list on the left and add them to the list on the right by using the arrows. You can also move the properties up or down to adjust the display order by using the arrows on the right side of the dialog.

TIP: You must select at least one property on the Properties tab to see data in the query output. In the ContentList rule, some properties are selected by default.

8 On the **Sort** tab, specify how the data will be sorted in the query output. For each property you select, you can specify the sort order (ascending or descending).

9 Once you've finished editing the action, click **Exit**.

10 To save your changes, click **Yes**.

11 Save the rule.

12 To test your query, add the ContentList portlet to a portal page and test the page.

V Reference

Describes how to use the Content Management (CM) JSP tag library

- Chapter 23, “Content Management Tag Library”

23 Content Management Tag Library

This chapter describes the tags in the Content Management (CM) tag library (**ContentMgmtTag.jar**):

📖 For background information, see the chapter on using the exteNd Director tag libraries in *Developing exteNd Director Applications*.

Content Management tags:

- checkIn
- checkOut
- findDocuments
- getChildDocuments
- getContent
- getDirectory
- getDirectoryList
- getDocType
- getDocument
- getFieldInfo
- getFields
- getLinkedDocuments
- getVersionHistory
- publish
- unCheckOut
- updateDocument

Alphabetical list of tags

checkIn

Description

Checks a document in to the CM subsystem for the current user and saves a new content version. If the save is successful, the tag returns an integer representing the new version.

This tag wraps the checkinDocument() method on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:checkIn docid="docID" mime="mime" content="content"
comment="comment" keepcheckedout="keepcheckedout" id="ID" />
```

Attribute	Required?	Request-time expression values supported?	Description
docid	Yes	Yes	Specifies the UUID for a document in the CM subsystem.
mime	Yes	Yes	Specifies the MIME type of the new version.
content	Yes	Yes	Specifies the new content data.
comment	No	Yes	Specifies a checkin comment.
keepcheckedout	Yes	Yes	Indicates whether the new version should be kept checked out. If true, the new version is inserted but the document remains checked out to the user. If false, the lock is released and the document is made available for changes by other users.
id	No	No	Specifies the name of the variable used to store the result of the operation. If the checkin is successful, this variable holds the new version. If no value is specified, a default id of **version** is used.

Example

```
<% taglib uri="/cm" prefix="cm" %>
...
<%
String content = "this is my new content";
byte myarray[] = content.getBytes();
%>
...
<cm:checkIn docid="addd2545931b11d48e130010a4e70c5f" id="version"
comment="checking in my changes" content="<%=myarray%>"
keepcheckedout="true" mime="text/html" />
<%=pageContext.getAttribute("version")%>
```

checkOut

Description

Checks out a document for the current user, returning true if successful and false if unsuccessful.

This tag wraps the checkoutDocument() method on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:checkOut docid="docID" id="ID" />
```

Attribute	Required?	Request-time expression values supported?	Description
docid	Yes	Yes	Specifies the UUID for a document in the CM subsystem.
id	No	No	Specifies the name of the variable that is used to store the result of the operation. If no value is specified, a default id of **checkout** is used.

Example

```
<% taglib uri="/cm" prefix="cm" %>
...
<cm:checkOut docid="addd2545931b11d48e130010a4e70c5f" id="result" />
<%=pageContext.getAttribute("result")%>
```

findDocuments

Description

Retrieves documents that match the criteria specified in tag attributes (as described below), returning either a list of EbiDocument objects or an XML string.

This tag wraps either the findElements() or the findFilteredElements() method of the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:findDocuments id="ID" secure="securitySetting" xml="xmlFormat"
authorFrom="authorFrom" authorTo="authorTo" authorLike="authorLike"
categoryID="categoryID" createDateFrom="createDateFrom"
createDateTo="createDateTo" expireDateFrom="expireDateFrom"
expireDateTo="expireDateTo" publishDateFrom="publishDateFrom"
publishDateTo="publishDateTo" docTypeName="docTypeName"
folderID="folderID" docNameFrom="docNameFrom" docNameTo="docNameTo"
docNameLike="docNameLike" parentDocId="parentDocID"
titleFrom="titleFrom" titleTo="titleTo" titleLike="titleLike"
orderAsc="orderAsc" orderDesc="orderDesc"/>
```

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable used to store the list of EbiDocument objects. If no value is specified, a default id of **foundDocuments** is used.
secure	No	No	Specifies whether the returned documents are filtered according to security constraints. If true (the default), the filter method is used and only those documents to which the user has read access are returned. If false, all documents are returned.

Attribute	Required?	Request-time expression values supported?	Description
xml	No	No	Specifies that the document list is returned as an XML string. The DTD for the returned xml is contentmgmt-query-results_3_0.dtd, which can be found under templates\Director\library\ContentMgmtService/ContentMgmtService-conf/DTD directory in the standard exteNd Director installation directory. If not specified, a list of EbiDocument objects is returned.
authorFrom authorTo	No	Yes	Search for documents based on a range of author metadata.
authorLike	No	Yes	Search for documents based on a match of author metadata. This attribute is case-insensitive, and may include SQL wildcard characters **%** and _.
categoryID	No	Yes	Limits the search to documents in a particular category.
createDate From createDateTo	No	Yes	Search for documents based on creation date metadata. Date entries should have the format **m/d/yyyy**. For example: **5/14/2001** is a valid date entry.
expireDate From expireDateTo	No	Yes	Search for documents based on expiration date metadata. Date entries should have the format **m/d/yyyy**. For example: **5/14/2001** is a valid date entry.
publishDate From publishDateTo	No	Yes	Search for documents based on publication date metadata. Date entries should have the format **m/d/yyyy**. For example: **5/14/2001** is a valid date entry.

Attribute	Required?	Request-time expression values supported?	Description
docTypeName	No	Yes	Limits the search to documents of a specific type.
folderId	No	Yes	Limits the search to documents in a particular folder.
docName From docNameTo	No	Yes	Search for documents based on a range of document name metadata.
docNameLike	No	Yes	Search for documents based on a match of document name metadata. This attribute is case-insensitive, and may include SQL wildcard characters % and _.
parentDocId	No	Yes	Limits the search to documents that are children of a particular document.
titleFrom titleTo	No	Yes	Search for documents based on a range of title metadata.
titleLike	No	Yes	Search for documents based on a match of title metadata. This attribute is case-insensitive, and may include SQL wildcard characters % and _.
orderAsc orderDesc	No	Yes	Sorts the documents in ascending or descending order, based on one of the search criteria. Legitimate values for these attributes are: ◆ author ◆ createDate ◆ docId ◆ docName ◆ expireDate ◆ publishDate ◆ title

Example

```
<cm:findDocuments id="test2" secure="false" xml="false"
       authorLike="administrator" orderAsc="DOCID" />
Found <%=test2.size()%> Documents <br/>
<% for (int x=0;x<test2.size();x++){

       EbiDocument doc = (EbiDocument) test2.get(x); %>
       Doc <%=x%> title = <%=doc.getTitle()%>

<% } %>
```

getChildDocuments

Description

Retrieves the children of a document, returning a list of EbiDocument objects.

Depending on the setting for the **secure** attribute, this tag wraps either the getChildDocuments() or the getFilteredChildDocuments() method on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:getChildDocuments docid="docID" docpath="docPath" id="ID"
secure="securitySetting"/>
```

Attribute	Required?	Request-time expression values supported?	Description
docid	No	Yes	Specifies the UUID for the parent document in the CM subsystem. If you do not specify a docid value, you must specify a value for the docpath attribute.
docpath	No	Yes	Specifies the path to the parent document in the CM subsystem. If you do not specify a docpath value, you must specify a value for the docid attribute.
id	No	No	Specifies the name of the variable used to store the returned list of EbiDocument objects. If no value is specified, a default id of **childDocuments** is used.

Attribute	Required?	Request-time expression values supported?	Description
secure	No	No	Specifies whether the returned documents are filtered according to security constraints. If true (the default), the filter method is used and only those documents to which the user has read access are returned. If false, all documents are returned.

Example

```
<cm:getChildDocuments docid="c373e9ea8d110d2c8f6a0000864ec468"
id="test3"/>
Found <%=test3.size()%> Child Documents <br/>
```

getContent

Description

Retrieves the contents of a document, returning a string.

This tag wraps the getContent() method on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:getContent docid="docID" docpath="docpath" id="ID"
version="version" verid="verID" />
```

Attribute	Required?	Request-time expression values supported?	Description
docid	No	Yes	Specifies the UUID for a document in the CM subsystem. If a docid value is not specified, you must specify a value for the docpath attribute.
docpath	No	Yes	Specifies the path to a document in the CM subsystem. If a docpath value is not specified, you must specify a value for the docid attribute.

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable used to store the EbiDocContent object. If no value is specified, the document content is inserted in the page at the location where the tag appears.
version	No	No	Indicates whether to return a specified version of the content. If false (the default), or if this attribute is omitted, the published version of the content is returned. If no version is published, no content is returned. If true, you must specify a version using the verid attribute. The specified version is returned.
verid	No	Yes	Specifies a version ID for the content that should be returned. If the version attribute is false, the verid attribute is ignored.

Example

This example gets the latest version of the content for two documents by specifying the paths to the documents. The content for each document is inserted in the page at the location where the corresponding getContent tag appears:

```
<% taglib uri="/cm" prefix="cm" %>
...
<cm:getContent docpath="HR/Employee Forms/ESPP/ChangeOfAddress.html"
id="doc1" />
content = <%=new String(((EbiDocContent)doc1).getData())%>
<cm:getContent docpath="HR/Employee
Forms/ESPP/ChangeOfAddressInstructions.html" id="doc2" />
content = <%=new String(((EbiDocContent)doc2).getData())%>
...
```

getDirectory

Description

Retrieves a directory from the CM subsystem, returning an EbiDirectory object. This tag can be used to retrieve folders as well as categories.

This tag wraps the getEntry() and lookupDirectoryEntry() methods on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:getDirectory id="id" roottype="roottype" dirname="dirname"
dirid="dirid" dirpath="dirpath" />
```

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable used to store the EbiDirectory object. If no value is specified, the default name of **dirEntry** is used for the variable.
roottype	Yes	Yes	Specifies whether the tag is being used to retrieve a folder or a category. If the directory is a folder, specify **folder** as the value for the roottype attribute. If it is a category, specify **category** instead. Typically, this attribute is used in conjunction with one of the following attributes to specify the correct directory object in the CM subsystem: ◆ dirname ◆ dirid ◆ dirpath If none of these attributes is specified, the root folder or category is returned, depending on the setting of roottype.
dirname	No	Yes	Specifies the name of the directory you want to retrieve. The directory specified must be a direct descendent of the root. The directory can be a folder or category in the CM subsystem.
dirid	No	Yes	Specifies the UUID for the directory you want to retrieve. The directory can be a folder or category in the CM subsystem.

Attribute	Required?	Request-time expression values supported?	Description
dirpath	No	Yes	Specifies the path to the directory you want to retrieve. The directory can be a folder or category in the CM subsystem.

Example

```
<%@ taglib uri="/cm" prefix="cm" %>
...
<cm:getDirectory roottype="category" dirpath="HR/Employee Forms/ESPP" >

ID for the directory is ...
<%=dirEntry.getID()%>
```

getDirectoryList

Description

Retrieves a list of directory contents from the CM subsystem, returning a collection of EbiDirectoryEntry objects. Depending on the attributes specified, this collection can contain folder, category, and document objects.

This tag wraps the getDirectoryList() and getFilteredDirectoryList() methods on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:getDirectoryList id="id" finddocuments="finddocuments"
roottype="roottype" parentdir="parentdir" iterate="iterate"
findsubdirs="findsubdirs" dirname="dirname" dirid="dirid"
dirpath="dirpath" filter="documents" />
```

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable used to store the Collection object. If a value is specified for the id attribute, that value is used as the name for the resulting variable that contains the collection. Otherwise, the default name of **dirList** is used for the variable.

Attribute	Required?	Request-time expression values supported?	Description
finddocuments	No	No	Indicates whether to retrieve the documents that are located in the specified directory. If true, all documents located in the specified directory are retrieved. If false (the default), documents located in the specified directory are not retrieved.
roottype	No	Yes	Specifies whether to retrieve the contents of a folder or a category. If the directory is a folder, specify **folder** as the value for the root attribute. If it's a category, specify **category**. Typically, this attribute is used in conjunction with one of the following attributes: ◆ dirname ◆ dirid ◆ dirpath If the roottype attribute is specified by itself, the directory for which contents will be retrieved is the root. If a value for this attribute is not specified, the directory for which contents will be retrieved is assumed to be the root folder.
parentdir	No	Yes	Specifies the directory object for which the document contents should be retrieved. The object should be of type EbiDirectory. If this attribute is specified, it is not necessary to specify the roottype attribute.

Attribute	Required?	Request-time expression values supported?	Description
iterate	No	No	Indicates whether this tag operates as a body tag so that each row can be processed separately. If true, the following values can be accessed within the getDirectoryList tag: ◆ identifier ◆ name ◆ type ◆ isdir Each of these variables has a scope of NESTED. If false (the default), this tag operates as a nonbody tag. The tag returns an object of type Collection that contains a collection of EbiDirectoryEntry objects.
findsubdirs	No	No	Indicates whether to retrieve directories that are child directories under the specified one. If true (the default), all subdirectories of the specified directory are retrieved. If false, subdirectories of the specified directory are not retrieved.
dirname	No	Yes	Specifies the name of a directory from which contents should be retrieved. The directory specified must be a direct descendent of the root. The directory can be a folder or category in the CM subsystem.

Attribute	Required?	Request-time expression values supported?	Description
dirid	No	Yes	Specifies the UUID for a directory from which contents should be retrieved. The directory can be a folder or category in the CM subsystem.
dirpath	No	Yes	Specifies the path to a directory from which contents should be retrieved. The directory can be a folder or category in the CM subsystem.
filter	No	No	Indicates whether to search using security filters. If true (the default), the filter method is used and only those objects to which the user has read access are returned. If false, all objects are returned.

Examples

This example shows how to use the getDirectoryList tag with the iterate attribute set to **true**:

```
<%@ taglib uri="/cm" prefix="cm" %>
...
<cm:getDirectoryList roottype="category" dirpath="HR/Employee
Forms/ESPP" finddocuments="true" iterate="true">
Identifier = <%=identifier%><br/>
Name = <%=name%><br/>
Type = <%=type%><br/>
Is this item a directory? = <%=isdir%><br/>
</cm:getDirectoryList>
```

This example shows how to use the getDirectoryList tag with the iterate attribute set to **false**:

```
<%@ taglib uri="/cm" prefix="cm" %>
...
<cm:getDirectoryList iterate="false" filter="false"/>

<%= ((java.util.List)pageContext.getAttribute("dirList")).size() %>
= the size of the list...
```

getDocType

Description

Retrieves a document type from the CM subsystem, returning an EbiDocType object.

Depending on whether you specify the typeid or name attribute, this tag wraps the getDocumentTypeByID() or getDocumentTypeByName() method on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:getDocType typeid="docID" name="name" id="ID" />
```

Attribute	Required?	Request-time expression values supported?	Description
typeid	No	Yes	Specifies the UUID for a document type in the CM subsystem. If you do not specify a typeid value, you must specify a value for the name attribute.
name	No	Yes	Specifies the name of a document type in the CM subsystem. If you do not specify a name value, you must specify a value for the typeid attribute.
id	No	No	Specifies the name of the variable used to store the EbiDocType object. If no value is specified, a default id of **docType** is used.

Example

```
<% taglib uri="/cm" prefix="cm" %>
...
<cm:getDocType typeid="addd2543931b11d48e130010a4e70c5f" id="test" />
...
Name for the doc type is ...
<%=test.getDocTypeName()%>
```

getDocument

Description

Retrieves a document, returning an EbiDocument object.

This tag wraps the lookupDirectoryEntry() and getDocument() methods on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:getDocument docid="docID" docpath="docPath" id="ID" />
```

Attribute	Required?	Request-time expression values supported?	Description
docid	No	Yes	Specifies the UUID for a document in the CM subsystem. If you do not specify a docid value, you must specify a value for the docpath attribute.
docpath	No	Yes	Specifies the path to a document in the CM subsystem. If you do not specify a docpath value, you must specify a value for the docid attribute.
id	No	No	Specifies the name of the variable used to store the EbiDocument object. If no value is specified, a default id of **document** is used.

Example

```
<% taglib uri="/cm" prefix="cm" %>
...
<cm:getDocument id="test" docid="addd2545931b11d48e130010a4e70c5f" />
Title for document is ...
<%=test.getTitle()%>
```

getFieldInfo

Description

Retrieves the extension fields of a document, returning the field information as an EbiDocExtnMeta object.

This tag wraps the getDocumentExtnMeta() method on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:getFieldInfo docid="docID" docpath="docPath" id="ID"
iterate="iterateSetting"/>
```

Attribute	Required?	Request-time expression values supported?	Description
docid	No	Yes	Specifies the UUID for a document in the CM subsystem. If you do not specify a docid value, you must specify a value for the docpath attribute.
docpath	No	Yes	Specifies the path to a document in the CM subsystem. If you do not specify a docpath value, you must specify a value for the docid attribute.
id	No	No	Specifies the name of the variable used to store the EbiDocExtnMeta object. If no value is specified, a default id of **docFields** is used.

Attribute	Required?	Request-time expression values supported?	Description
iterate	No	No	Indicates whether this tag operates as a body tag so that each row can be processed separately. If true, the following values can be accessed within the getDirectoryList tag: ◆ fieldInfo ◆ fieldName ◆ fieldValues Each of these variables has a scope of NESTED. If false (the default), this tag operates as a nonbody tag. The tag returns an EbiDocExtnMeta object.

Example

```
<cm:getFieldInfo docid="c373e9ea8d110d2c8f6a0000864ec468" id="test6" />
<% for (int x=0;x<test6.size();x++){

        EbiDocExtnMetaInfo dmi = (EbiDocExtnMetaInfo) test6.get(x); %>
        Field <%=x%> info = <%=dmi.getFieldName()%>

<% } %>
```

getFields

Description

Retrieves fields from the CM subsystem, returning a collection of EbiDocField objects. You can use this tag to retrieve all fields or fields for a given document type.

This tag wraps the getDocumentFields() and getFilteredDocumentFields() methods on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:getFields id="ID" doctypeid="doctypeID"
doctypename="doctypename" iterate="iterate" filter="filter" />
```

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable used to store the collection of EbiDocField objects. If no value is specified, a default name of **fieldList** is used for the variable.
doctypeid	No	Yes	Specifies the UUID for a document type in the CM subsystem. If you do not specify either a doctypeid or doctypename value, all fields are retrieved.
doctypename	No	Yes	Specifies the name of a document type in the CM subsystem. If you do not specify either a doctypeid or doctypename value, all fields are retrieved.
iterate	No	No	Indicates whether this tag is to operate as a body tag so that each row can be processed separately. If true, the following values can be accessed within the getFields tag: ◆ identifier ◆ name Each of these variables has a scope of NESTED. If false (the default), this tag operates as a nonbody tag. In this case, the tag returns a collection of EbiDocField objects.
filter	No	No	Indicates whether to search using security filters. If true (the default), the filter method is used and only those fields to which the user has read access are returned. If false, all fields are returned.

Examples

This example shows how to use the getFields tag with the iterate attribute set to **true**:

```
<%@ taglib uri="/cm" prefix="cm" %>
...
<cm:getFields doctypename="myDocumentType" iterate="true">
Identifier = <%=identifier%><br/>
Name = <%=name%><br/>
</cm:getFields>
```

This example shows how to use the getFields tag with the iterate attribute set to **false**:

```
<%@ taglib uri="/cm" prefix="cm" %>
...
<cm:getFields iterate="false" filter="false"/>

<%= ((java.util.List)pageContext.getAttribute("fieldList")).size()
%> = the size of the list...
```

getLinkedDocuments

Description

Retrieves the documents linked to a particular document, returning a list of EbiDocument objects.

Depending on the attributes you specify for this tag, it wraps one of these methods on the EbiContentMgmtDelegate interface:

- getFilteredLinkChildDocuments()
- getLinkChildDocuments()
- getFilteredLinkParentDocuments()
- getLinkParentDocuments()

Syntax

```
<prefix:getLinkedDocuments docid="docID" docpath="docPath" id="ID"
secure="securitySetting" parentLinks="parentLinksSetting"/>
```

Attribute	Required?	Request-time expression values supported?	Description
docid	No	Yes	Specifies the UUID for a document in the CM subsystem. If you do not specify a docid value, you must specify a value for the docpath attribute.

Attribute	Required?	Request-time expression values supported?	Description
docpath	No	Yes	Specifies the path to a document in the CM subsystem. If you do not specify a docpath value, you must specify a value for the docid attribute.
id	No	No	Specifies the name of the variable used to store the list of EbiDocument objects. If no value is specified, a default id of **linkedDocuments** is used.
secure	No	No	Specifies whether the returned documents are filtered according to security constraints. If true (the default), the filter method is used and only those documents to which the user has read access are returned. If false, all documents are returned.
parentLinks	No	No	Specifies whether you want to get documents that are linked as parents or children to the specified document. If true, return parent documents to which this document is linked. If false (the default), return child documents that are linked to this document.

Example

```
<cm:getLinkedDocuments docid="c373e9ea8d110d2c8f6a0000864ec468"
       id="test4" parentLinks="false"/>
Found <%=test4.size()%> Linked Documents <br/>
```

getVersionHistory

Description

Retrieves the versions of a document, returning a list of EbiDocVersion objects.

This tag wraps the getDocumentContentVersions() method on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:getVersionHistory docid="docID" docpath="docPath" id="ID"
includeContent="includeContent"/>
```

Attribute	Required?	Request-time expression values supported?	Description
docid	No	Yes	Specifies the UUID for a document in the CM subsystem. If you do not specify a docid value, you must specify a value for the docpath attribute.
docpath	No	Yes	Specifies the path to a document in the CM subsystem. If you do not specify a docpath value, you must specify a value for the docid attribute.
id	No	No	Specifies the name of the variable used to store the list of EbiDocVersion objects. If no value is specified, a default id of **docVersions** is used.
include Content	No	No	Include the actual content in the returned EbiDocVersion objects.

Example

```
<cm:getVersionHistory docid="c373e9ea8d110d2c8f6a0000864ec468"
       id="test1" includeContent="false"/>
<% EbiDocVersion ver = (EbiDocVersion) test1.get(0); %>
Version mime-type is = <%=ver.getMimeType()%>
```

publish

Description

Publishes a specified version of content for a document, returning true if successful or false if unsuccessful.

This tag wraps the publishDocumentContentVersion() method on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:publish docid="docID" uselatest="uselatest" version="version"
overwrite="overwrite" force="force" />
```

Attribute	Required?	Request-time expression values supported?	Description
docid	Yes	Yes	Specifies the UUID for a document in the CM subsystem.
uselatest	Yes	No	Indicates whether to publish the latest version of the document. If true, the latest version is published. If false, the version number specified in the version attribute is published.
version	No	Yes	Specifies the version to publish. This attribute is required if the uselatest attribute is set to false.
overwrite	No	No	Indicates whether to replace any versions already published. If true (the default), the specified version overwrites any published version for the document. If false, an exception is thrown if a published version of the document already exists.
force	No	No	Indicates whether to force an immediate publish, regardless of the publish dates specified in the document metadata. If true, the version is published regardless of the publish date value specified for the document. If false (the default), the current data and time is compared against the publish date and time specified for the document. If it is too early or too late to publish, the version is not published; otherwise, the version is published. In either case, an application exception is thrown.

Example

```
<% taglib uri="/cm" prefix="cm" %>
...
<cm:publish docid="addd2545931b11d48e130010a4e70c5f" uselatest="true"
/>
```

unCheckOut

Description

Unchecks out a document from the CM subsystem for the current user, returning true if successful or false if unsuccessful.

NOTE: No data is saved. Any changes made between the original checkout and the uncheckout are lost.

This tag wraps the unCheckOutDocument() method on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:unCheckOut docid="docID" id="ID" />
```

Attribute	Required?	Request-time expression values supported?	Description
docid	Yes	Yes	Specifies the UUID for a document in the CM subsystem.
id	No	No	Specifies the name of the variable used to store the result of the operation. If no value is specified, a default id of **uncheckout** is used.

Example

```
<% taglib uri="/cm" prefix="cm" %>
...
<cm:unCheckOut docid="addd2545931b11d48e130010a4e70c5f" id="done" />
<%=pageContext.getAttribute("done")%>
```

updateDocument

Description

Updates a document in the CM subsystem, returning true if successful or false if unsuccessful.

This tag wraps the updateDocument() method on the EbiContentMgmtDelegate interface.

Syntax

```
<prefix:updateDocument doc="document" checkout="checkoutSetting"
checkin="checkinSetting"/>
```

Attribute	Required?	Request-time expression values supported?	Description
doc	No	Yes	Specifies a document (an object of class EbiDocument) in the CM subsystem.
checkout	No	Yes	Specifies that the document is to be checked out to the current user before performing the update.
checkin	No	Yes	Specifies that the document is to be checked in after performing the update.

Example

```
<cm:getDocument id="test" docid="c373e9ea8d110d2c8f6a0000864ec468" />
...
<% test.setAbstract(test.getAbstract()+"a"); %>
<cm:updateDocument doc="<%=test%>" checkout="true" checkin="true"/>
```

Index

A

access
  permissions in Content Management subsystem 79
  restricting 82
access control
  in Content Management subsystem 77
access right types
  and Content Management subsystem 79
ACLs
  about 33
  access methods for ContentAdmin 82
  access methods in Content Management subsystem 80
  adding (code examples) 82
  inheriting 81
  specifying for new objects 81
administrator
  ContentAdmin 80
  role 78
attachments
  adding 238
author role 78
auto-checkin feature for documents
  about 200
  enabling 264
Auto Create utility for Content Management subsystem 224, 229
auto-publish feature for documents
  about 200, 266
  enabling 268

B

browsers
  identifier strings 41

C

cascading security in CMS Administration Console 283
categories
  assigning to documents 246
  creating 219
  deleting 254
  managing 43
category parameter 25
category tree 46
child documents
  adding 64, 237
  getting 68
  updating link 67
cleanup data feature for document types 202, 258
CMS Administration Console
  about 24
  accessing 192
  administering tasks 301
  Auto Create utility 224
  classifying content 25
  content list 195
  content tree view 195
  content view tabs 194
  context-sensitive toolbar 195
  creating content 221
  interactive controls 194
  main page 193
  Property Inspector 195
  tasks 189
  toolbar 194
  using the internal HTML Editor 229
cm.sssw.cm.api package 30
compound document relationship 62
compound linking
  about 65
  methods for 66
content, in Content Management subsystem
  about display styles 24
  classifying 25
  compared to pages 21
  creating 221
  default formats in document types 200
  defined 20, 221
  defining structure and layout 24
  deleting 253
  dynamic 223

editing 244
exporting 110
exporting in the CMS Administration Console 289
importing 112
importing in the CMS Administration Console 293
previewing 242
securing 77
versions of 26
ContentAdmin group 80
ContentList
sample application 307
ContentList rule
editing 308
Content Management subsystem
API 30
auto-publish feature 266
changing data about content 30
checking documents in and out 260
creating and adding fields 202
creating categories 219
creating display styles 212
creating documents 223
creating document types 199
creating folders 198
creating relationships between documents 236
creating tasks 90
creating taxonomies 218
customizing tasks 89
default document type 24
document types, about 24
getting manager (code example) 30
installed tasks 86
layout styles, adding 41
logical infrastructure 23
physical infrastructure 22
publishing document versions 264
removing relationships between documents 239
repository 30
rolling back document versions 265
security for 77
security methods 80
setting security on content elements 284
system document type 255
tasks, managing 85
tasks, overview of 31
unpublishing document versions 265
users, roles for 78
version control 264
Content Query
sample application 307
Content Query action
in Content Query sample application 308
content security
access permissions 281
cascading 283
permissions required for Content Management tasks 282
setting 284
control types
for document fields 202

D

data export descriptor (DED)
about 113
samples 113
data import descriptor (DID)
about 114
sample 114
data types
about 32
default task 86
display styles
about 24
creating 212, 214
deleting 255
modifying 249
document fields
see fields
documents
adding 49
adding category (code example) 45
adding child (code example) 64
adding fields 34
adding layout document 42
auto-checkin feature 200
auto-publish feature 200
categories (code example) 46
changing layout style 43
checking in 263
checking in and out 260, 261, 263
child document, updating (code example) 67
child documents, getting (code example) 68
composite 74
compound document relationship 62
creating 223
creating relationships between 236
defined 20
deleting 254

displaying 72
extension metadata 55
fields by name 55
fields for document (code example) 57
fields for type (code example) 54
field values, geting 57
field values, setting 55
hierarchical relationship 62
HTML, setting in a portlet 72
HTML content, displaying 72
layout sets 60
layout styles, managing 38
managing folders and categories 43
metadata for 32, 49
methods for managing 58
modifying and publishing 69
parent, getting (code example) 68
publishing a version 267
removing relationships between 239
rolling back to a previous version 268
setting expiration dates 252
status, setting (code example) 71
status of 70
style sheets 38
types, managing 35
unpublishing a version 267
when to check out 260
XML content, displaying 73
XML layout, getting (code example) 73
document types
about 24
adding 37
cleanup data feature 202, 258
creating 199
default 24
deleting 255, 256
editing 251
system 255

E

EbiContentMgmtDelegate 80
elements
securable in Content Management subsystem 79
events
Content Management types 115
enabling in Content Management 123
enabling task 103
enabling WebDAV 185
in Content Management 115
registering in Content Management 119, 121
registering WebDAV 184
tasks and 102
types, in WebDAV 183
types, specifiying in Content Management 120
WebDav and 183
exceptions
handling (code example) 84
expire task 86
export behavior in the CMS Administration Console 288
exporting content
about 289
API 114
customizing 113
process overview 111
extension metadata
and documents 55
see also fields

F

fields
adding 34
adding with document type 37
and document types 33
and values 33
creating and adding 202
deleting from document types 257
deleting from the CMS Administration Console 257
document (code example) 55
document type (code example) 54
editing 252
getting by name 55
getting for document 57
getting for document (code example) 57
getting values for (code example) 57
managing 32
setting values in document 55
system 255
values, getting 57
values for a document, specifying 54
folders
assigning to documents 246
creating 198
deleting 253
managing 43
specifying for documents 228

G

groups
in Content Management subsystem 78

H

hierarchical document relationship 62
hierarchical linking 63
HTML Editor
using in the CMS Administration Console 229
hyperlinks
creating in documents 233

I

images
inserting in documents 235
import behavior in the CMS Administration Console 288
importing content
about 293
API 114
customizing 113
process overview 112

J

janitor task 86
JavaScript
writing in the CMS Administration Console 206

L

layout document
adding 42
layout styles
adding 41
adding layout document descriptor 42
changing 43
layout sets 60
managing 38
setting up 39
legacy documents
removing fields from parent document types 258
linking
checking out target documents 65
compound 65
hierarchical 63
logical content infrastructure in CMS Administration Console 23

M

manager objects
getting for content (code example) 30
metadata
for documents 32, 49
for fields 33
list of predefined elements 223
methods
for managing documents 58
for version control and publishing 71
modes, in the CMS Administration Console 194

P

pages
compared to CMS Administration Console content 21
parent documents
getting 68
permissions
and Content Management subsystem elements 79
for ContentAdmin group 80
physical content infrastructure in CMS Administration Console 22
_PmcSystemDefaultType for content management 255, 257
portlets
HTML, setting in 72
properties
modifying 245
Property Inspector
exporting content from 292
publishing
dates 269
methods for 71
publisher role 78
publish task 86, 264

R

repository (Content Management)
changing information about 30
restricted access
in Content Management subsystem 82
roles
in Content Management subsystem 78
rolling back
defined 265

S

securable objects
accessing ACLs for 80
access right types 79
in Content Management subsystem 79
security
cascading 283
for Content Management tasks 282
for the Content Management subsystem 77
permissions for content access 281
setting on content elements 284
security exceptions
handling (code example) 84
style sheets
and CMS Administration Console display styles 24, 212
and Content Management subsystem 38, 41
creating 213
how managed by CMS Administration Console 212
layout sets for content 60
uploading to CMS Administration Console 215
synch task 86
system administrator
ContentAdmin 80
system document fields 255
system document type 255

T

tag libraries
Content Management tag library 315
tasks
administering in the CMS Administration Console 301
creating 90
customizing 89
default 86
events and 102
expire 86
installed 86
janitor 86
managing 85
publish 86
registering and configuring 87
synch 86
taxonomies
assigning to documents 246
creating 218
deleting 254
text
copying 233
cutting 233
formatting 233
pasting 233

U

unpublish task
defined 265
user agents 41
users
in Content Management subsystem 78

V

version control
auto-checkin feature 200, 264
auto-publish feature 200, 266
in Content Management subsystem 264
methods for 71
publishing 264
rolling back 265
unpublishing 265
views, in the CMS Administration Console 194

W

WebDAV client
about 137
adding a category reference to a document 150
classes 141
configuring your environment 138
constructing WebDAV requests that use Proppatch 146

deleting a document using a helper method (code example) 144
deleting a document using utility methods (code example) 147
helper methods 141
how exteNd Director manages versioning for 131
how exteNd Director secures content from 131
how exteNd Director stores content from 130
issuing WebDAV requests from a Java client 149, 150
programming practices using helper methods 144
programming practices using utility methods 146
setting up 133
using 139
utility methods 142
WebDAV requests and responses 140
why build your own 138
working with resources, collections, and properties 140
WebDAV client, issuing WebDAV requests from a Java client
adding a category reference using a helper method (code example) 150
adding a category reference using utility methods (code example) 151
copying a document using a helper method (code example) 154
copying a resource or collection 154
creating a collection using a helper method (code example) 155
creating a document using a helper method (code example) 157
creating a new collection 155
creating a new document from a custom template 156
deleting a document 158
getting a document using utility methods (code example) 158
getting allowed methods using utility methods (code example) 162
getting a resource or collection 158
getting header information from a resource or collection 160
getting header information using utility methods (code example) 160
getting methods that can be called on a resource or collection 162
getting properties defined on a resource or collection 164
getting properties using utility methods (code example) 164
locking a document 166
locking a document using a helper method (code example) 166
moving a folder using a helper methods (code example) 168
moving a resource or collection 168
removing a category reference from a document 169
removing a category reference using a helper method (code example) 169
removing a category reference using utility methods (code example) 170
removing all category references from a document 172
removing all category references using a helper method (code example) 172
removing all category references using utility methods (code example) 173
renaming a document using a helper method (code example) 175
renaming a resource or collection 175
setting a field value using a helper method (code example) 177
setting a field value using utility methods (code example) 178
setting the value of a custom field in a document 177
unlocking a document 180
unlocking a document using a helper method (code example) 180
updating a document 181
updating a document using a helper method (code example) 181
WebDAV protocol
about 127
and distributed Web authoring 128
extensions to HTTP 128
WebDAV subsystem
about 127
deploying 133
events and 183
exteNd Director support for 129
installing 132
supported methods 134
what you can do 130

X

XML
document categories (code example) 46
getting for a document 73