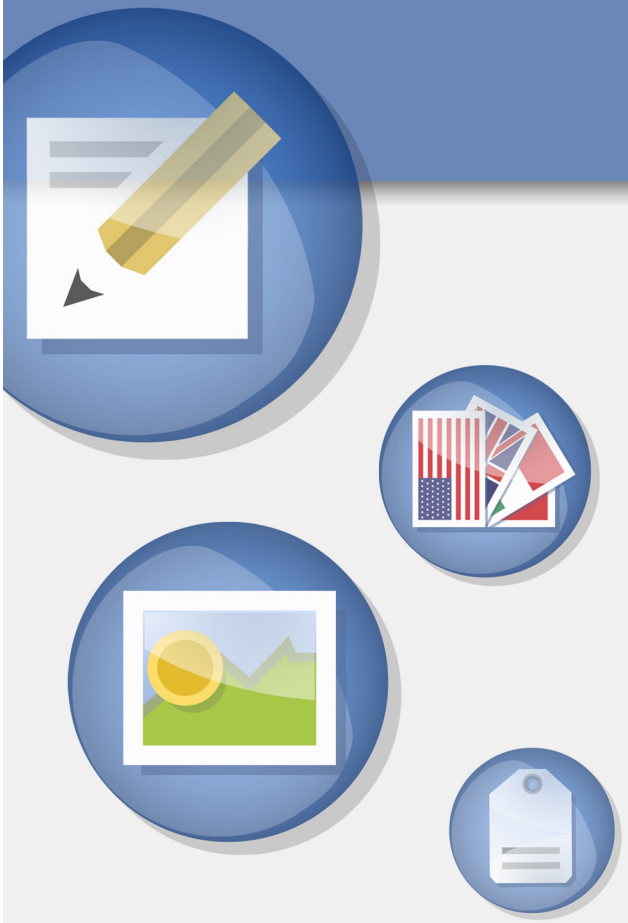
# FatWire | Content Server 7

Version 7.6 Patch 2



## Developer's Guide

**Document Revision Date:** Jan. 31, 2012

FatWire®
SOFTWARE

FATWIRE CORPORATION PROVIDES THIS SOFTWARE AND DOCUMENTATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. In no event shall FatWire be liable for any direct, indirect, incidental, special, exemplary, or consequential damages of any kind including loss of profits, loss of business, loss of use of data, interruption of business, 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 or the documentation even if FatWire has been advised of the possibility of such damages arising from this publication. FatWire may revise this publication from time to time without notice. Some states or jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

Copyright © 2012 FatWire Corporation. All rights reserved.

The release described in this document may be protected by one or more U.S. patents, foreign patents or pending applications.

FatWire, FatWire Content Server, FatWire Engage, FatWire Satellite Server, CS-Desktop, CS-DocLink, Content Server Explorer, Content Server Direct, Content Server Direct Advantage, FatWire InSite, FatWire Analytics, FatWire TeamUp, FatWire Content Integration Platform, FatWire Community Server and FatWire Gadget Server are trademarks or registered trademarks of FatWire, Inc. in the United States and other countries.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. AIX, AIX 5L, WebSphere, IBM, DB2, Tivoli and other IBM products referenced herein are trademarks or registered trademarks of IBM Corporation. Microsoft, Windows, Windows Server, Active Directory, Internet Explorer, SQL Server and other Microsoft products referenced herein are trademarks or registered trademarks of Microsoft Corporation. Red Hat, Red Hat Enterprise Linux, and JBoss are registered trademarks of Red Hat, Inc. in the U.S. and other countries. Linux is a registered trademark of Linus Torvalds. SUSE and openSUSE are registered trademarks of Novell, Inc., in the United States and other countries. XenServer and Xen are trademarks or registered trademarks of Citrix in the United States and/or other countries. VMware is a registered trademark of VMware, Inc. in the United States and/or various jurisdictions. Firefox is a registered trademark of the Mozilla Foundation. UNIX is a registered trademark of The Open Group in the United States and other countries. Any other trademarks and product names used herein may be the trademarks of their respective owners.

This product includes software developed by the Indiana University Extreme! Lab. For further information please visit http://www.extreme.indiana.edu/.

Copyright (c) 2002 Extreme! Lab, Indiana University. All rights reserved.

This product includes software developed by the OpenSymphony Group (http://www.opensymphony.com/).

The OpenSymphony Group license is derived and fully compatible with the Apache Software License; see http://www.apache.org/LICENSE.txt.

Copyright (c) 2001-2004 The OpenSymphony Group. All rights reserved.

You may not download or otherwise export or reexport this Program, its Documentation, or any underlying information or technology except in full compliance with all United States and other applicable laws and regulations, including without limitations the United States Export Administration Act, the Trading with the Enemy Act, the International Emergency Economic Powers Act and any regulations thereunder. Any transfer of technical data outside the United States by any means, including the Internet, is an export control requirement under U.S. law. In particular, but without limitation, none of the Program, its Documentation, or underlying information of technology may be downloaded or otherwise exported or reexported (i) into (or to a national or resident, wherever located, of) any other country to which the U.S. prohibits exports of goods or technical data; or (ii) to anyone on the U.S. Treasury Department's Specially Designated Nationals List or the Table of Denial Orders issued by the Department of Commerce. By downloading or using the Program or its Documentation, you are agreeing to the foregoing and you are representing and warranting that you are not located in, under the control of, or a national or resident of any such country or on any such list or table. In addition, if the Program or Documentation is identified as Domestic Only or Not-for-Export (for example, on the box, media, in the installation process, during the download process, or in the Documentation), then except for export to Canada for use in Canada by Canadian citizens, the Program, Documentation, and any underlying information or technology may not be exported outside the United States or to any foreign entity or “foreign person” as defined by U.S. Government regulations, including without limitation, anyone who is not a citizen, national, or lawful permanent resident of the United States. By using this Program and Documentation, you are agreeing to the foregoing and you are representing and warranting that you are not a “foreign person” or under the control of a “foreign person.”

*FatWire Content Server Developer’s Guide*
Document Revision Date: Jan. 31, 2012
Product Version: 7.6 Patch 2

**FatWire Headquarters**
FatWire Corporation
330 Old Country Road
Suite 303
Mineola, NY 11501

Table of

# Contents

**About This Guide . . . . . 25**
Who Should Use This Guide . . . . . 25
How This Guide Is Organized . . . . . 25
Related Documents . . . . . 26
Conventions . . . . . 26
Third-Party Libraries . . . . . 26

## Part 1. Overview

**1 Overview of FatWire Content Server . . . . . 29**
Content Server Product Family . . . . . 30
Product Summary . . . . . 30
Third-Party Components . . . . . 34
J2EE Compliance . . . . . 34
Content Server Systems . . . . . 35
The Content Server Core . . . . . 36
Servlets and Java APIs . . . . . 36
Page-Generation Components . . . . . 39
Database Management Functions . . . . . 41
Sessions and Cookies . . . . . 42
Event Management Features . . . . . 43
Satellite Server . . . . . 43
Handling the HTTP Requests . . . . . 43
Satellite Server Servlets and APIs . . . . . 44
CS-Direct . . . . . 44
Basic Asset Model . . . . . 45
The Advanced Interface . . . . . 48
The Dash Interface . . . . . 50
Content Server Clients (Interface Options) . . . . . 50
Sites and the Site Plan . . . . . 51

Sample Sites . . . . . 51
Template, CSElement, and SiteEntry Assets . . . . . 52
Custom CS-Direct XML and JSP Tags. . . . . 52
Approval and Publishing. . . . . 52
Revision Tracking . . . . . 53
Workflow . . . . . 53
Searching and Search Engines . . . . . 54
CS-Direct Advantage. . . . . 54
Flex Asset Model . . . . . 54
Assetsets and Searchstates: Searching the Online Site . . . . . 57
Shopping Carts and Commerce Context. . . . . 57
GE Lighting Sample Site . . . . . 58
Custom CS-Direct Advantage XML and JSP tags . . . . . 58
Content Server Clients (Interface Options) . . . . . 58
Engage . . . . . 60
Visitor Data and Segments . . . . . 60
Recommendations. . . . . 61
Promotions . . . . . 61
Persistent, Linked Visitor Sessions. . . . . 61
Custom Engage XML and JSP tags . . . . . 61
Content Server Portal Interface . . . . . 62

**2 Overview of Sites . . . . . 63**
Content Management Sites . . . . . 64
Online Sites . . . . . 65
Developers and the Content Management Site . . . . . 66
Sites and the Site Plan . . . . . 66
Sites and the Database . . . . . 68

**3 Content Server Development Process. . . . . 69**
Step 1: Set Up the Team . . . . . 70
Step 2: Create Functional and Design Specifications . . . . . 71
Functional Requirements . . . . . 71
Page Design. . . . . 71
Caching Strategy . . . . . 71
Security Strategy (Access Control). . . . . 71
Separate Format from Content (Elements from Assets) . . . . . 72
Data Design. . . . . 73
Step 3: Set Management System Requirements . . . . . 74
Step 4: Implement the Data Design . . . . . 75
Step 5: Build the Online Site . . . . . 75
Step 6: Set Up the Management System . . . . . 76
Import Content as Assets . . . . . 76
Import Catalog Data and Flex Asset Data . . . . . 76
Instruct the Editorial Team About Site Design. . . . . 76

Step 7: Set Up the Delivery System . . . . . 77
Step 8: Publish to the Delivery System . . . . . 77

## Part 2. Programming Basics

**4 Programming with Content Server . . . . . 81**
Choosing a Coding Language . . . . . 82
The Content Server Context . . . . . 82
The ICS Object . . . . . 82
The FTCS tag . . . . . 83
Content Server JSP . . . . . 83
Content Server Standard Beginning . . . . . 83
JSP Implicit Objects . . . . . 86
Syntax . . . . . 86
Actions . . . . . 86
Declarations . . . . . 87
Scriptlets and Expressions . . . . . 87
JSP Directives . . . . . 87
Content Server Tag Libraries . . . . . 87
Content Server XML . . . . . 90
Content Server Standard Beginning . . . . . 90
XML Entities and Reserved Characters . . . . . 91
XML Parsing Errors . . . . . 91
Content Server Tags . . . . . 92
Tags That Create the Content Server Context . . . . . 93
Tags That Handle Variables . . . . . 93
Tags That Call Pages and Elements . . . . . 94
Tags That Create URLs. . . . . 95
Tags That Control Caching . . . . . 96
Tags That Set Cookies . . . . . 96
Programming Construct Tags . . . . . 97
Tags That Manage Compositional and Approval Dependencies . . . . . 97
Tags That Retrieve Information About Basic Assets . . . . . 98
Tags That Create Assetsets (Flex Assets) . . . . . 100
Tags That Create Searchstates (Flex Assets) . . . . . 102
Variables . . . . . 104
Reserved Variables . . . . . 104
Setting Regular Variables . . . . . 106
Setting Session Variables . . . . . 107
Working With Variables . . . . . 109
Variables and Precedence . . . . . 112
Best Practices with Variables . . . . . 112
Other Content Server Storage Constructs . . . . . 113
Built-ins. . . . . 113

Lists . . . 113
Counters . . . 114
Values for Special Characters . . . 115

**5 Page Design and Caching . . . 117**
Modular Page Design . . . 118
Caching . . . 119
Content Server Caching . . . 119
BlobServer and Caching . . . 119
Satellite Server Caching . . . 120
Viewing the Contents of the Satellite Server Cache . . . 125
CacheManager . . . 126
The SiteCatalog Table . . . 126
The Cache Key . . . 126
Caching Properties . . . 127
Double-Buffered Caching . . . 128
Implementing Double-Buffered Caching . . . 131
Setting cscacheinfo . . . 133
Coding for Caching . . . 133
Caching and Security . . . 134

**6 Intelligent Cache Management with Content Server . . . 137**
Content Server's Rendering Engine Cache . . . 138
CacheManager . . . 138
Enabling CacheManager . . . 139
Tier 1 Cache Configuration Properties . . . 139
Tier 2 Cache Configuration Properties . . . 141

**7 Advanced Page Caching Techniques . . . 143**
Overview . . . 144
Configuring the Content Server Cache . . . 144
Configuring the Blob Server Cache . . . 147
Configuring the Satellite Server Cache . . . 147
CacheInfo String Syntax . . . 148
CacheInfo String: First Part . . . 148
CacheInfo String: Second Part . . . 149
Page Timeout . . . 149
Absolute Moment in Time . . . 149
TimePattern . . . 149
Wildcard . . . 150
Blank . . . 150

**8 Content Server Tools and Utilities . . . 151**
Content Server Explorer . . . 152
Connecting to a Content Server Database . . . 152

CatalogMover . . . . . 153
Starting CatalogMover . . . . . 153
Connecting to Content Server . . . . . 154
CatalogMover Menu Commands . . . . . 154
Exporting Tables . . . . . 156
Importing Tables . . . . . 158
Command Line Interface . . . . . 160
Property Editor . . . . . 161
Starting the Property Editor . . . . . 161
Setting Properties . . . . . 162
Merging Property Files . . . . . 162
XMLPost . . . . . 163

**9 Sessions and Cookies . . . . . 165**
What Is a Session? . . . . . 166
Session Lifetime . . . . . 166
Session Variables Maintained by Content Server . . . . . 166
Logging In and Logging Out . . . . . 167
Sessions Example . . . . . 167
FeelingsForm Element . . . . . 167
SetFeeling Element . . . . . 168
Meat Element . . . . . 168
What Is a Cookie? . . . . . 169
CookieServer . . . . . 169
Cookie Tags . . . . . 169
Cookie Example . . . . . 170
Start.xml . . . . . 170
ColorForm . . . . . 171
CreateCookie . . . . . 171
DisplayWelcome . . . . . 171
Running the Cookie Example . . . . . 172
Tips and Tricks . . . . . 172
Satellite Server Session Tracking . . . . . 172
Flushing Session Information . . . . . 172

**10 Error Logging and Debugging . . . . . 175**
Overview . . . . . 176
Logging to Content Server's Log File . . . . . 176
Using Error Codes with Tags . . . . . 177
Tag Examples Using Error Codes . . . . . 177
Error Number Rules . . . . . 178

# Part 3. Data Design

**11 Data Design: The Asset Models . . . . . 181**
Asset Types and Asset Models . . . . . 182
Two Data Models . . . . . 182
Default (Core) Asset Types. . . . . 182
Which Asset Model Should You Use to Represent Your Content? . . . . . 185
The Basic Asset Model . . . . . 186
Basic Asset Types from the Burlington Financial Sample Site . . . . . 186
Relationships Between Basic Assets. . . . . 187
Category, Source, and Subtype . . . . . 188
Basic Asset Types and the Database. . . . . 190
The Flex Asset Model . . . . . 195
The Flex Family . . . . . 196
Sample Site Flex Families. . . . . 197
Flex Attributes. . . . . 198
Flex Parents and Flex Parent Definitions . . . . . 200
Flex Assets and Flex Definition Assets . . . . . 202
Flex Families and the Database . . . . . 203
Assetsets and Searchstates. . . . . 206
Search Engines and the Two Asset Models. . . . . 207
Tags and the Two Asset Models . . . . . 207
Summary: Basic and Flex Asset Models. . . . . 208
Where the Asset Models Intersect . . . . . 208
Where the Asset Models Differ . . . . . 208
Summary: Asset Types . . . . . 209

**12 The Content Server Database . . . . . 211**
Types of Database Tables . . . . . 212
Object Tables . . . . . 212
Tree Tables . . . . . 213
Content Tables . . . . . 214
Foreign Tables. . . . . 214
System Tables . . . . . 215
Identifying a Table's Type . . . . . 216
Types of Columns (Fields) . . . . . 217
Generic Field Types . . . . . 217
Database-Specific Field Types . . . . . 218
Indirect Data Storage with the Content Server URL Field . . . . . 219
Creating Database Tables . . . . . 220
Creating Object Tables . . . . . 220
Creating Tree Tables. . . . . 222
Creating Content Tables . . . . . 223
Registering a Foreign Table . . . . . 225
How Information Is Added to the System Tables . . . . . 226

Property Files and Databases . . . . . 227

**13 Managing Data in Non-Asset Tables . . . . . 229**
Methods and Tags . . . . . 230
Writing and Retrieving Data . . . . . 230
Querying for Data . . . . . 232
Lists and Listing Data . . . . . 232
Coding Data Entry Forms . . . . . 234
Adding a Row . . . . . 234
Deleting a Row . . . . . 237
Querying a Table . . . . . 240
Querying a Table with an Embedded SQL Statement . . . . . 248
Managing the Data Manually . . . . . 252
Deleting Non-Asset Tables . . . . . 253

**14 Resultset Caching and Queries . . . . . 255**
Overview . . . . . 256
Caching Frameworks . . . . . 256
Database Queries . . . . . 256
How Resultset Caching Works . . . . . 256
Reducing the Load on the Database . . . . . 257
Specifying the Table Name . . . . . 257
SELECTTO . . . . . 258
EXECSQL . . . . . 258
CALLSQL . . . . . 258
Search Forms in the Content Server Interface . . . . . 258
Query Asset . . . . . 259
SEARCHSTATE . . . . . 259
Flushing the Resultset Cache . . . . . 259
Enabling Resultset Caching over inCache . . . . . 260
Configuring Resultset Caching . . . . . 260
Table-Specific Properties . . . . . 261
Planning Your Resultset Caching Strategy . . . . . 261
Summary . . . . . 262

**15 Designing Basic Asset Types . . . . . 263**
The AssetMaker Utility . . . . . 264
How AssetMaker Works . . . . . 264
Asset Descriptor Files . . . . . 268
Columns in the Asset Type’s Database Table . . . . . 270
Elements and SQL Statements for the Asset Type . . . . . 274
Creating Basic Asset Types . . . . . 278
Overview . . . . . 278
Before You Begin . . . . . 279
Step 1: Code the Asset Descriptor File . . . . . 280

Step 2: Upload the Asset Descriptor File to Content Server. . . . . 288
Step 3: Create the Asset Table . . . . . 289
Step 4: Configure the Asset Type . . . . . 290
Step 5: Enable the Asset Type on Your Site. . . . . 291
Step 6: Fine-Tune the Asset Descriptor File. . . . . 291
Step 7: (Optional) Customize the Asset Type Elements. . . . . 292
Step 8: (Optional) Configure Subtypes. . . . . 295
Step 9: (Optional) Configure Association Fields . . . . . 296
Step 10: (Optional) Configure Categories . . . . . 298
Step 11: (Optional) Configure Sources. . . . . 299
Step 12: (Conditional) Add Mimetypes . . . . . 300
Step 13: (Optional) Edit Search Elements to Enable Indexed Search . . . . . 301
Step 14: Create and Assign Asset Type Icons (Dash Interface only). . . . . 301
Step 15: Code Templates for the Asset Type . . . . . 302
Step 16: Move the Asset Types to Other Systems . . . . . 302
Deleting Basic Asset Types. . . . . 302
Images and eWebEditPro . . . . . 303

**16 Designing Flex Asset Types . . . . . 305**
Design Tips for Flex Families . . . . . 306
Visitors on the Delivery System . . . . . 306
Users on the Management System . . . . . 306
How Many Attribute Types Should You Create?. . . . . 307
Designing Flex Attributes . . . . . 307
How Many Definition Types Should You Create?. . . . . 309
Designing Parent Definition and Flex Definition Assets . . . . . 309
Summary . . . . . 311
The Flex Family Maker Utility . . . . . 311
The Flex Asset Elements. . . . . 311
Creating a Flex Asset Family . . . . . 312
Overview. . . . . 312
Before You Begin . . . . . 313
Step 1: Create a Flex Family. . . . . 313
Step 2: (Optional) Create Additional Flex Family Members . . . . . 315
Step 3: Enable the New Flex Asset Types . . . . . 316
Step 4: Create Flex Attributes. . . . . 317
Step 5: (Optional) Create Flex Filter Assets . . . . . 321
Step 6: Create Parent Definition Assets . . . . . 323
Step 7: Create Flex Definition Assets. . . . . 324
Step 8: Create Flex Parent Assets . . . . . 326
Step 9: Create and Assign Asset Type Icons (Dash Interface only). . . . . 328
Step 10: Code Templates for the Flex Assets . . . . . 328
Step 11: Test Your Design (Create Test Flex Assets) . . . . . 328
Step 12 (Optional): Create Flex Asset Associations. . . . . 328
Step 13: Move the Asset Types to Other Systems . . . . . 329

Editing Flex Attributes, Parents, and Definitions . . . . . 330
Editing Attributes . . . . . 330
Editing Parent Definitions and Flex Definitions. . . . . 330
Editing Parents and Flex Assets . . . . . 331
Using Product Sets. . . . . 332
What Is a Product Set? . . . . . 332
Creating Product Sets . . . . . 332

**17 Flex Filters . . . . . 333**
Overview . . . . . 334
Flex Filter Classes . . . . . 335
Flex Filter Assets . . . . . 337
Defining a Flex Filter Class and Creating a Flex Filter Asset. . . . . 337
Implementing a Flex Filter Class . . . . . 338
Defining a Custom Flex Filter Class. . . . . 340
Creating Flex Filter Assets . . . . . 341
Document Transformation . . . . . 346
Default Solution . . . . . 346
Custom Solutions . . . . . 346
Customizing Document Transformation . . . . . 347
Next Steps . . . . . 350

**18 Designing Attribute Editors . . . . . 351**
Overview . . . . . 352
The presentationobject.dtd File . . . . . 353
The Attribute Editor Asset . . . . . 355
The Attribute Editor Elements . . . . . 361
Conventions for the Attribute Editor Elements. . . . . 363
Creating Attribute Editors . . . . . 365
Customizing Attribute Editors . . . . . 367
Example: Customized Attribute Editor. . . . . 367
Editing Attribute Editors . . . . . 372

**19 Configuring Bundled Attribute Editors . . . . . 373**
Configuring FCKEditor . . . . . 374
Before You Begin . . . . . 374
Creating an FCKEditor Instance and Enabling It for a Field . . . . . 374
Enabling FCKEditor for Use in the InSite Interface. . . . . 376
Implementing Lazyload . . . . . 377
Enabling Selected Asset Types for the FCKEditor. . . . . 378
Setting the Approval Dependency for Included Assets . . . . . 379
Enable “Image Picker” for the FCKEditor . . . . . 380
Customizing the FCKEditor Toolbar . . . . . 381
Configuring Spell Check Support in FCKEditor . . . . . 383
Enabling Custom Style Sheets for FCKEditor . . . . . 383

Creating and Testing Style Sheets . . . . . 384
Configuring the Clarkii Online Image Editor . . . . . 392
Creating a Clarkii OIE Instance and Enabling it for a Field . . . . . 393
Configuring Clarkii OIE Properties . . . . . 396
Implementing a Field Copier Filter to Classify Assets . . . . . 399
Enabling Clarkii OIE for Use in the InSite Interface . . . . . 402
Configuring the Image Picker . . . . . 403
Categorizing Image Assets for Display in Image Picker . . . . . 403
Sample Image Picker Attribute Editor Definition Code . . . . . 404
Configuring eWebEditPro . . . . . 404
Configuring the RENDERFLASH Attribute Editor . . . . . 405
Sample RENDERFLASH Attribute Editor Definition Code . . . . . 406

**20 Importing Assets of Any Type . . . . . 407**
The XMLPost Utility . . . . . 408
Overview . . . . . 408
XMLPost Configuration Files . . . . . 410
Configuration Properties for XMLPost . . . . . 410
Configuration Properties for the Posting Element . . . . . 413
Configuration Properties for the Source Files . . . . . 415
Sample XMLPost Configuration File . . . . . 419
XMLPost Source Files . . . . . 421
Sample XMLPost Source File . . . . . 421
XMLPost and File Encoding . . . . . 421
Using the XMLPost Utility . . . . . 422
Before You Begin . . . . . 422
Running XMLPost from the Command Line . . . . . 422
Running XMLPost as a Batch Process . . . . . 425
Running XMLPost Programmatically . . . . . 425
Customizing RemoteContentPost and PreUpdate . . . . . 426
Setting a Field Value Programmatically . . . . . 426
Setting an Asset Association . . . . . 427
Troubleshooting XMLPost . . . . . 428
Debugging the Posting Element . . . . . 428

**21 Importing Flex Assets . . . . . 429**
Overview . . . . . 430
Importing the Data Structure Flex Asset Types . . . . . 430
Importing the Flex Assets . . . . . 430
Importing Flex Assets: The Process . . . . . 430
Using Custom Data Delimiters . . . . . 432
XMLPost and the Flex Asset Model . . . . . 433
Internal Names vs. External Names . . . . . 433
Importing the Structural Asset Types in the Flex Model . . . . . 434
Attribute Editors . . . . . 434

Flex Attributes. . . . . 436
Flex Definitions and Flex Parent Definitions . . . . . 440
Flex Parents. . . . . 444
Importing Flex Assets with XMLPost. . . . . 445
Configuration File Properties and Source File Tags for Flex Assets . . . . . 446
Sample Flex Asset Configuration File for addData . . . . . 449
Sample Flex Asset Source File for addData . . . . . 451
Sample Flex Asset Configuration File for RemoteContentPost . . . . . 454
Sample Flex Asset Source File for RemoteContentPost. . . . . 455
Editing Flex Assets with XMLPost . . . . . 457
Configuration Files for Editing Flex Assets . . . . . 457
Source Files for Editing Flex Assets. . . . . 457
Deleting Assets with XMLPost . . . . . 459
Configuration Files for Deleting Assets . . . . . 459
Source Files for Deleting Assets. . . . . 460

**22 Importing Flex Assets with the BulkLoader Utility . . . . . 461**
Overview of BulkLoader . . . . . 462
BulkLoader Features. . . . . 462
How BulkLoader Works . . . . . 462
Using the BulkLoader Utility . . . . . 463
Importing Flex Assets from Flat Tables . . . . . 464
Step 1: Use XMLPost to Import Structural Assets . . . . . 465
Step 2: Create the Input Table (Data Source) . . . . . 465
Step 3: Create the Mapping Table. . . . . 467
Step 4: Create the BulkLoader Configuration File . . . . . 468
Step 5: Run the BulkLoader Utility . . . . . 475
Step 6: Review Feedback Information . . . . . 476
Step 7: Approve and Publish the Assets to the Delivery System . . . . . 476
Importing Flex Assets Using a Custom Extraction Mechanism . . . . . 476
IDataExtract Interface . . . . . 477
IPopulateDataSlice . . . . . 481
IFeedback Interface. . . . . 485
Approving Flex Assets with the BulkApprover Utility . . . . . 486
Configuring BulkApprover. . . . . 486
Using BulkApprover. . . . . 488

## Part 4. Site Development

**23 Creating Template, CSElement, and SiteEntry Assets. . . . . 493**
Overview . . . . . 494
Pages, Pagelets, and Elements . . . . . 495
Elements, Pagelets, and Caching . . . . . 495
Calling Pages and Elements . . . . . 495

Page vs. Pagelet . . . . . 497
CSElement, Template, and SiteEntry Assets . . . . . 497
Template Assets . . . . . 498
CSElement Assets . . . . . 499
SiteEntry Assets . . . . . 500
What About Non-Asset Elements? . . . . . 500
Creating Template Assets . . . . . 501
Prerequisites . . . . . 501
Procedures for Creating Template Assets . . . . . 505
Creating CSElement Assets . . . . . 521
Prerequisites . . . . . 521
Procedures for Creating CSElement Assets . . . . . 522
Creating SiteEntry Assets . . . . . 531
Prerequisites . . . . . 531
Procedures for Creating SiteEntry Assets . . . . . 532
Managing Template, CSElement, and SiteEntry Assets . . . . . 537
Designating Default Approval Templates (Static Publishing Only) . . . . . 537
Editing Template, CSElement, and SiteEntry Assets . . . . . 537
Sharing Template, CSElement, and SiteEntry Assets . . . . . 538
Deleting Template, CSElement, and SiteEntry Assets . . . . . 538
Previewing Template, CSElement, and SiteEntry Assets . . . . . 539
Using Content Server Explorer to Create and Edit Element Logic . . . . . 540
Creating Templates and CSElements . . . . . 541
Editing Templates and CSElements . . . . . 541

**24 Creating Templates to Support Graphical Page Design . . . . . 543**
Overview . . . . . 544
Implementation . . . . . 544
Template Context . . . . . 548
Guidelines for Creating Master Templates . . . . . 549
Tracking Changes to Master Pages . . . . . 550

**25 Creating Collection, Query, Stylesheet, and Page Assets . . . . . 553**
Previewing Assets . . . . . 554
Approving Assets . . . . . 554
Sharing Assets . . . . . 555
Deleting Assets . . . . . 556
Collection Assets . . . . . 556
Before You Begin . . . . . 556
Creating Collection Assets . . . . . 557
Sharing Collection Assets . . . . . 558
Query Assets . . . . . 558
Query Assets and Other Assets . . . . . 558
How the Query Is Stored . . . . . 559
Commonly Used Fields for Queries . . . . . 559

Before You Begin . . . . . 561
Creating Query Assets . . . . . 562
Sharing Query Assets . . . . . 563
Previewing and Approving Query Assets . . . . . 563
Stylesheet Assets . . . . . 564
Creating Stylesheet Assets . . . . . 564
Sharing Stylesheet Assets . . . . . 565
Page Assets . . . . . 565
Creating a Page Asset . . . . . 566
Placing Page Assets . . . . . 569
Moving Page Assets in the Site Tree . . . . . 570
Placing Page Assets and Workflow . . . . . 572
Editing Page Assets . . . . . 572
Deleting Page Assets . . . . . 572

**26 Best Practices for Creating Site Preview Assets and Templates . . . . . 573**
Considerations when Implementing Site Preview . . . . . 574
Creating Sets of Assets . . . . . 574
Writing Templates for Site Preview . . . . . 574
The asset:filterassetsbydate tag . . . . . 575
The Input List . . . . . 575
Caching Considerations . . . . . 577

**27 Coding Elements for Templates and CSElements . . . . . 579**
About Dependencies . . . . . 580
The Publishing System and Approval Dependencies . . . . . 580
Page Generation and Compositional Dependencies . . . . . 584
About Coding to Log Dependencies . . . . . 585
ASSET.LOAD and asset:load . . . . . 586
The ASSETSET (assetset) Tag Family . . . . . 586
RENDER.GETPAGEURL and render:getpageurl . . . . . 587
RENDER.LOGDEP (render:logdep) . . . . . 587
RENDER.FILTER and render:filter . . . . . 588
RENDER.UNKNOWNDEPS and render:unknowndeps . . . . . 589
Calling CSElement and SiteEntry Assets . . . . . 589
Coding Elements to Display Basic Assets . . . . . 590
Assets That Represent Simple Content . . . . . 591
Associations . . . . . 592
ImageFile Assets or Other Blob Assets . . . . . 593
Basic Assets That Can Have Embedded Links . . . . . 593
Collections . . . . . 594
Query Assets . . . . . 595
Page Assets . . . . . 597
About Coding Elements that Display Flex Assets . . . . . 599
Assetsets . . . . . 599

Searchstates . . . . . 600
Assetsets, Searchstates, and Flex Attribute Asset Types . . . . . 601
Scope . . . . . 601
Coding Templates That Display Flex Assets . . . . . 602
Example Data Set for the Examples in This Section . . . . . 602
Examples of Assetsets with One Product (Flex Asset) . . . . . 603
Special Cases: Flex Attributes of Type Text, Blob, and URL . . . . . 605
Examples of Assetsets with More Than One Product (Flex Asset) . . . . . 608
Creating URLs for Hyperlinks . . . . . 613
RENDER.GETPAGEURL (render:getpageurl) . . . . . 613
RENDER.SATELLITEBLOB (render:satelliteblob) . . . . . 614
RENDER.GETBLOBURL (render:getbloburl) . . . . . 614
Using the referURL Variable . . . . . 615
Handling Error Conditions . . . . . 616
Using the Errno Variable . . . . . 616
Ensuring that Incorrect Pages Are Not Cached . . . . . 617

**28 Template Element Examples for Basic Assets . . . . . 619**
Example 1: Basic Modular Design . . . . . 620
First Element: Home . . . . . 621
Second Element: MainStoryList . . . . . 621
Third Element: LeadSummary . . . . . 623
Fourth Element: TeaserSummary . . . . . 623
Back to LeadSummary . . . . . 624
Back to MainStoryList . . . . . 624
Back to Home . . . . . 625
Example 2: Coding Links to the Article Assets in a Collection Asset . . . . . 625
First element: SectionFront . . . . . 625
Second element: PlainList . . . . . 626
Example 3: Using the ct Variable . . . . . 627
First Element: SectionFront . . . . . 628
Second Element: TextOnlyLink . . . . . 629
ColumnistFront . . . . . 630
Example 4: Coding Templates for Query Assets . . . . . 631
First Element: Home . . . . . 631
Second Element: WireFeedBox . . . . . 632
Third Element: ExecuteQuery . . . . . 633
Back to WireFeedBox . . . . . 633
Example 5: Displaying an Article Asset Without a Template . . . . . 634
First Element: Full . . . . . 634
Second Element: AltVersionBlock . . . . . 635
Third Element: EmailFront . . . . . 635
Example 6: Displaying Site Plan Information . . . . . 636
First Element: Home . . . . . 636
Second Element: SiteBanner . . . . . 636

Third Element: TopSiteBar . . . . . 637
Back to SiteBanner . . . . . 639
Example 7: Displaying Non-Asset Information . . . . . 639
First Element: Home . . . . . 639
Second Element: ShowMainDate . . . . . 639

**29 Configuring Sites for Multilingual Support . . . . . 641**
Overview . . . . . 642
Dimensions . . . . . 642
Dimension Sets . . . . . 642
Cross-Site Multilingual Support . . . . . 643
Master Assets, Translations, and Multilingual Sets . . . . . 644
Translations and Asset Relationships . . . . . 645
Approval Dependencies . . . . . 646
Working with Locale Filtering . . . . . 647
Handling Asset Relationships Through Locale Filtering . . . . . 647
Included Locale Filters . . . . . 648
Custom Locale Filters . . . . . 650
Compositional Dependencies . . . . . 650
Adding Filtering Support to Your Site . . . . . 651
Planning Multilingual Support for a Site . . . . . 653
Configuring Multilingual Support for a Site . . . . . 654
Configuration Quick Reference . . . . . 654
Enabling the “Dimension” and “DimensionSet” Asset Types . . . . . 655
Enabling the “Locale” Subtype of the “Dimension” Asset Type . . . . . 656
Creating a Locale . . . . . 656
Sharing a Locale to Another Site . . . . . 657
Creating and Configuring a Dimension Set . . . . . 658
Sharing a Dimension Set to Another Site . . . . . 659
Configuring a Locale Filter . . . . . 659
Configuring the Fallback Hierarchy of the Hierarchical Filter . . . . . 661
Bulk-Assigning a Default Locale to Assets in a Site . . . . . 662

**30 Setting Up Flash Content Management . . . . . 665**
Overview . . . . . 666
The Flash Management Model . . . . . 666
Configuring Flash Support . . . . . 667
How Flash Content Is Rendered . . . . . 668
Prerequisites for Setting Up Flash Content Support . . . . . 668
Configuring Flash Content Support on a Site . . . . . 670
Sample Element Code for Rendering Flash Assets . . . . . 674
The Generated <OBJECT>/<EMBED> Tag . . . . . 679

**31 User Management on the Delivery System . . . . . 681**
The Directory Services API . . . . . 682

Entries . . . . . 682
Hierarchies . . . . . 682
Groups . . . . . 682
Directory Services Tags . . . . . 683
Directory Operations . . . . . 683
Error Handling . . . . . 686
Troubleshooting Directory Services Applications . . . . . 687
Controlling Visitor Access to Your Online Sites . . . . . 688
ACL Tags . . . . . 688
User Tags . . . . . 689
Content Server and Encryption . . . . . 689
Creating Login Forms . . . . . 689
Prompt for Login (PromptForLogin.xml) . . . . . 689
Root Element for the Login Page . . . . . 690
Creating User Account Creation Forms . . . . . 691
PromptForNewAccount . . . . . 691
Root Element for the CreateAccount Page . . . . . 692
Visitor Access in the Burlington Financial Sample Site . . . . . 695
Membership Table . . . . . 695
Users and Passwords . . . . . 695
Member Accounts . . . . . 695
Membership Processing Elements . . . . . 695

**32 The HelloAssetWorld Sample Site . . . . . 697**
Overview . . . . . 698
HelloAssetWorld Templates . . . . . 698
HelloAssetWorld Asset Types . . . . . 699
Modified Asset Types . . . . . 699
The HelloArticle Asset Type . . . . . 699
The HelloImage Asset Type . . . . . 700
HelloAssetWorld Templates . . . . . 702
The HelloArticle Template . . . . . 702
The HelloCollection Template . . . . . 705
The HelloPage Template . . . . . 708
The HelloQuery Asset . . . . . 710

**33 The Burlington Financial Sample Site . . . . . 711**
Overview . . . . . 712
Navigation Features . . . . . 713
Breadcrumbs . . . . . 714
Best Practices . . . . . 715
Searching . . . . . 715
Keywords . . . . . 715
Hot Topics . . . . . 715
Topic Directory . . . . . 716

Related Stories . . . . . 717
Text-Only Versions . . . . . 718
Plain Text Parallel Site . . . . . 718
E-mail This Story . . . . . 719
AssetMaker Asset Types . . . . . 719
Mimetype . . . . . 719
Collections of Collections . . . . . 720
Membership . . . . . 720
Wire Feed . . . . . 721
Featured Funds . . . . . 721
Fund Finder . . . . . 721
Page Cache Parameters . . . . . 722

## Part 5. Management System Features

**34 Customizing the User Interface . . . . . 725**
Overview of the Tree . . . . . 726
Loading the Tree Tabs . . . . . 727
Refreshing the Tree . . . . . 734
Trees and Security . . . . . 735
Tree Error Logging . . . . . 735

**35 Coding for the InSite Editor . . . . . 737**
Overview . . . . . 738
The INSITE.EDIT Tag . . . . . 740
Parameters . . . . . 740
Syntax . . . . . 741
Supported Data Types and Input Types . . . . . 742
Template Element Examples . . . . . 742
Example for Basic Asset . . . . . 743
Example for Flex Assets . . . . . 744
Example for an Attribute of Type Blob . . . . . 745

**36 Customizing Workflow . . . . . 747**
Workflow Step Conditions . . . . . 748
Workflow Actions . . . . . 749
Step Action Elements . . . . . 750
Timed Action Elements . . . . . 752
Deadlock Action Elements . . . . . 754
Group Deadlock Action Elements . . . . . 757
Delegation Action Elements . . . . . 759

## Part 6. Web Services

**37 Overview of Web Services . . . . . 763**
What Are Web Services? . . . . . 764
SOAP and Web Services . . . . . 764
Supported SOAP Version . . . . . 765
Supported WSDL Version . . . . . 765
Related Programming Technologies . . . . . 765

**38 Creating and Consuming Web Services . . . . . 767**
Using Predefined Web Services . . . . . 768
Accessible Information . . . . . 768
WSDL File Location . . . . . 768
Process Flow for Predefined Web Services . . . . . 768
Consider Your Data for Predefined Web Services . . . . . 769
Generating the Client Interface . . . . . 769
Writing Client Calls . . . . . 769
Creating Custom Web Services . . . . . 769
Process Flow for Creating Custom Web Services . . . . . 770
Consider Your Data for Custom Web Services . . . . . 770
Creating a Content Server Page . . . . . 770
Writing a Content Server Element . . . . . 771
Creating a WSDL File . . . . . 773

## Part 7. Engage

**39 Creating Visitor Data Assets . . . . . 777**
About Visitor Data Assets . . . . . 778
Visitor Attributes . . . . . 778
History Attributes and History Definitions . . . . . 778
Segments . . . . . 778
Categories . . . . . 779
Developing Visitor Data Assets: Process Overview . . . . . 779
Creating Visitor Data Assets . . . . . 780
Creating Visitor Attributes . . . . . 780
Creating History Attributes . . . . . 783
Creating History Definitions . . . . . 786
Verifying Your Visitor Data Assets . . . . . 787
Approving Visitor Data Assets . . . . . 788

**40 Recommendation Assets . . . . . 789**
Overview . . . . . 790
Development Process . . . . . 790
Creating a Dynamic List Element . . . . . 791

**41 Coding Engage Pages . . . . . 793**
Commerce Context and Visitor Context . . . . . 794
Identifying Visitors and Linking Sessions. . . . . 794
Collecting Visitor Data . . . . . 795
Coding Site Pages That Collect Visitor Data . . . . . 796
Templates and Recommendations . . . . . 798
Creating Templates for Recommendations. . . . . 799
Shopping Carts and Engage. . . . . 800
Debugging Site Pages . . . . . 800
Session Links . . . . . 800
Visitor Data Collection . . . . . 801
Recommendations and Promotions. . . . . 801

**42 Memory-Centric Visitor Tracking . . . . . 803**
Overview . . . . . 804
Database-centric Model . . . . . 804
Memory-centric Model . . . . . 804
Enabling Memory-centric Visitor Tracking . . . . . 805
Visitor Tracking Property . . . . . 805
Supporting Code . . . . . 805
Batch-Saving History Attributes to the Database . . . . . 806
How Memory-centric Visitor Tracking Works . . . . . 806
1. Visitor Detection . . . . . 807
2. Retrieving Scalar Values. . . . . 808
3. Collecting History Attribute Values . . . . . 809
4. Computing Sums and Counts . . . . . 810
5. Computing Segments . . . . . 811
6. Determining and Displaying Recommended Assets. . . . . 812
Logging Dependencies . . . . . 813

## Part 8. API Reference

**43 Asset API . . . . . 817**
Overview . . . . . 818
Primary Interfaces . . . . . 818
Getting Started. . . . . 819
Asset API Read . . . . . 819
A Simple Example: Reading Field Values . . . . . 819
Reading AssetId . . . . . 820
Reading Attributes Given the Asset ID. . . . . 821
Query. . . . . 822
Complex Query . . . . . 824
Sorting. . . . . 825
Reading BlobObject . . . . . 825

Retrieving Multi-Valued Attributes . . . . . 826
Multilingual Assets: Retrieving Translations . . . . . 827
Reading Asset and Attribute Definitions . . . . . 828
Reading Key-Value Mappings . . . . . 828
Asset API Write . . . . . 830
Creating New Assets . . . . . 830
Updating Existing Assets . . . . . 831
Deleting Existing Assets . . . . . 832
Multilingual Assets . . . . . 832
Development Strategies . . . . . 834
Data Types and Attribute Data . . . . . 834
Query Types . . . . . 834
Optional: Setting Up to Use the Asset API from Standalone Java Programs . . . . . 837

**44 Public Site Search . . . . . 839**
Overview of the Search Framework . . . . . 840
Index Types . . . . . 841
Global Index . . . . . 841
Asset Type Index . . . . . 842
Search API Overview . . . . . 845
SearchEngine . . . . . 845
QueryExpression . . . . . 845
Configuration . . . . . 846
Basics of the Search API . . . . . 846
Advanced Configuration . . . . . 848
Configuring Lucene Parameters . . . . . 848
Configuring the Custom AnalyzerFactory . . . . . 851

## Appendices

**A. Creating a Hierarchical Flex Family . . . . . 855**
Overview . . . . . 856
Hierarchical Organization . . . . . 856
Flex Family Specifications . . . . . 857
Procedures . . . . . 858
Step 1: Create a Flex Family . . . . . 858
Step 2: Enable the New Flex Asset Types . . . . . 859
Step 3: Add a “Flex Family” Tab to Content Server’s Tree . . . . . 860
Step 5: Create Parent Definition Assets . . . . . 861
Step 6: Create Flex Parent Assets . . . . . 863
Step 7: Create Flex Definition Assets . . . . . 866
Step 8: Create Flex Assets . . . . . 870
Step 9: Translate the Formulaic Data Model into a Real-World Data Model . . . . . 873

Step 10: Develop Your Real-World Model . . . . . 875
Suggestions and Guidelines for Creating a Multi-Valued Model. . . . . 877
Next Steps . . . . . 878

**B. Content Server URL Assemblers . . . . . 879**
Overview of Content Server URL Assemblers . . . . . 880
URL Assembly . . . . . 880
Assembler Discovery and Disassembly . . . . . 880
URL Assembly and Disassembly Using GET and POST Requests . . . . . 880
Assemblers Installed with Content Server. . . . . 881
Working with Assemblers . . . . . 882
Creating Assemblers . . . . . 882
Registering and Ranking Assemblers . . . . . 882
Modifying Link Tags . . . . . 883

**C. White Space and Compression . . . . . 885**
White Space and JSP . . . . . 886
White Space and XML . . . . . 886
Compression . . . . . 886
JSP Design. . . . . 887

**D. Asset and Publish Events in Content Server . . . . . 889**
Overview . . . . . 890
Types of Events . . . . . 890
Asset Events . . . . . 891
Writing an Asset Event Listener . . . . . 891
Registering an Asset Event Listener . . . . . 891
Publishing Events . . . . . 892
Writing a Publishing Event Listener. . . . . 892
Registering a Publishing Event Listener. . . . . 893

**Index . . . . . 895**

# About This Guide

This guide describes the FatWire Content Server developer's environment. It begins with an overview of Content Server, its add-on products, and the development process you will follow to create your content management (CM) framework. The rest of the guide explains your main tasks:

- Building the online site.

  This requires designing and writing code to deliver content with the "look and feel" that best represents your organization's business. It also involves implementing page caching, security, and session management techniques.

- Creating the back end of the online site.

  This requires building content management (CM) sites, developing the data model to be used by those sites, and enabling optional interfaces for the end users. When the CM sites are ready for use, authorized content providers can work at those sites to create and manage content for the online site, collaborate in workflows, and publish to the online site.

- Implementing optional functionality; for example, web services and features that are provided by add-on products such as Engage.

## Who Should Use This Guide

This guide is written primarily for developers. It is assumed that developers have a clear knowledge of their company's business needs, and a basic understanding of their roles in the development of the online site and its back end. This guide is also useful to administrators, who collaborate with developers by setting up content management sites, site users, workflow processes, publishing methods, and Content Server client options.

Developers must know Java, JavaServer Pages (JSP), XML, and HTML. Administrators are not required to have programming experience, although a technical background is assumed.

## How This Guide Is Organized

Information in this guide is organized by parts, where each part presents a set of chapters that are related to a particular task or function.

Part 1, "Overview" presents an overview of Content Server, its add-on products, and the development process you will follow to create your content management (CM) framework.

Part 2, "Programming Basics" describes Content Server's programming environment—page caching techniques, tools and utilities for developing and maintaining online sites, session management options, and error logging and debugging techniques.

Part 3, "Data Design" presents information regarding data design, modeling, and management using the basic and flex asset models.

Part 4, "Site Development" explains how to develop your online site to deliver formatted content. It presents information about creating templates and other assets that make content delivery possible.

Part 5, "Management System Features" describes how to customize the Content Server interface and workflow processes.

Part 6, "Web Services" explains how to integrate Content Server with client applications that have a SOAP interface.

Part 7, "Engage" explains how to use this marketing product to gather demographic information and use that information for personalizing each visitor's page.

Part 8, "API Reference" explains how to use how to use Content Server's Asset API to perform read-write operations and support searches on the visitors' site.

The final part, "Appendices," provides tutorials showing how to create a hierarchically organized flex family, work with URL assemblers, implement whitespace compression techniques, and create event listeners.

## Related Documents

As you read this guide, keep a copy of the *Content Server Property Files Reference* handy, as well as a copy of the *Content Server Administrator's Guide*. The *Property Files Reference* provides detailed descriptions of the properties that are mentioned in this guide. The *Content Server Administrator's Guide* covers functions related to the management of CM sites, users, workflow processes, Content Server clients, and security.

## Conventions

The following text conventions are used in this guide:

- **Boldface** type indicates graphical user interface elements that you select.
- *Italic* type indicates book titles, emphasis, or variables for which you supply particular values.
- `Monospace` type indicates file names, URLs, sample code, or text that appears on the screen.
- **`Monospace bold`** type indicates a command.

## Third-Party Libraries

FatWire Content Server 7.6 patch 2 and its applications include third-party libraries. For additional information, see *FatWire Content Server 7.6 Patch 2: Third-Party Licenses*.

Part 1

# Overview

This part provides an overview of the Content Server products, and the development process. It contains the following chapters:


- Chapter 1, “Overview of FatWire Content Server”
- Chapter 2, “Overview of Sites”
- Chapter 3, “Content Server Development Process”

Chapter 1

# Overview of FatWire Content Server

The FatWire Content Server product family is a high-performance, large-scale content management and delivery system. You and your development team use the Content Server product family to create and manage large and complex web sites, portals, sites that run businesses, and other sites such as WAP, all of which are generically referred to as "online sites."

The Content Server product family is described in the following sections:

- Content Server Product Family
- The Content Server Core
- Satellite Server
- CS-Direct
- CS-Direct Advantage
- Content Server Clients (Interface Options)
- Engage
- Content Server Portal Interface

# Content Server Product Family

The Content Server product family consists of the Content Server product and several add-on offerings that support e-business constructs such as departmental sites, customer/partner web sites, and commerce implementations. This release also introduces support for the portal environment and offers a new add-on—the Content Server Portal interface.

The login screen of your CS system lists which products are installed on the system. Descriptions of the products are given in this section. Also included is a brief description of the supporting third-party software and CS compliance with J2EE standards.

## Product Summary

The **Content Server** product is the application on which the CS product family is built. Content Server is composed of the Content Server core, a set of developer's utilities, Satellite Server co-resident, two modules, and several clients that provide optional content management interfaces, mostly for business users.

- **Content Server core** is the operating system that all content applications are built upon. The Content Server core consists of APIs whose functions are to power the entire CS product family, communicate with the database that is chosen to store content, write content to and read content from the database, publish that content from system to system, and serve that content to site visitors. The Content Server core also provides basic page caching and query results caching functionality for enhanced system performance.

  While there is no graphical user interface, the Content Server core does provide **Content Server utilities**, which are the developer's tools for managing the Content Server database and various code. The utilities are GUI-based and must be manually installed (unless otherwise noted). They are:

  - CS-Explorer, for viewing and editing tables in the Content Server database. (This utility is automatically installed with the Content Server core.)
  - CatalogMover, for exporting and importing database tables.
  - XMLPost, for incrementally importing data into the Content Server database.
  - BulkLoader, for quickly importing large amounts of data into the Content Server database.
  - Property Editor, for viewing and organizing property files (system configuration files).

  Typically, e-businesses develop their content management systems not on the CS core, but on the CS module or product that content providers will use to produce and manage content for the online site. However, you always have the option to use the Content Server core alone to develop your own applications, if your e-business has special needs that the Content Server product line does not meet.

- **Satellite Server** (co-resident) is a caching application. It supplements Content Server's caching functionality by providing additional page caches. The tandem use of the CS and Satellite Server caches results in automatic double-buffered caching, ensuring that outdated content is never displayed on your live web site.

  Satellite Server is installed by default with Content Server. The same Satellite Server application is available as a separate product for installation on remote Content Server systems.

- **Content Server Direct (CS-Direct)** is the base module, built on the Content Server core.

  CS-Direct introduces the **basic asset model**, in which content entered by CS users is stored as objects called **assets** in the Content Server database. Each type of asset is contained in one primary storage table in the database, such that basic assets of one type can be associated with basic assets of another type. However, the assets cannot inherit each other's properties (called "attributes" in this guide). The basic asset model thus supports only flat data structures.

  Using Java APIs in the Content Server core, CS-Direct renders the Advanced, Dash, and InSite interfaces, allowing easy access to functions in the Content Server core and providing functions to support the basic asset model.

  CS-Direct also supports several clients that deliver alternative interfaces. The clients are described in the bulleted item "Content Server clients," on this page.

- **Content Server Direct Advantage (CS-Direct Advantage)** is the advanced data module. It is built on top of the Content Server core and on top of CS-Direct in order to make use of their functionality as well as the interface.

  Independently, CS-Direct Advantage introduces a comprehensive data model, called the **flex asset model**, in which each asset type uses several storage tables such that hierarchical data structures can be created, and child assets inherit attribute values from their parent assets. The flex asset model also supports flat data structures, within its own framework. (Note that the flex asset model functions independently of the basic asset model; tables created within the two models do not intersect.)

  Whether you choose the flex asset model or the basic asset model depends on the complexity of the data you plan to serve to your visitors. The flex asset model has historically been used for creating large online catalogs of products. However, it can be used in less complex situations, and is especially desirable when the intent is to eventually convert flat data structures to hierarchical structures. The conversion process does not require you to re-create the data.

  CS-Direct Advantage also supports e-commerce applications by introducing commerce-related features such as the shopping cart construct.

- **Content Server clients** are optional interfaces, mostly for the content providers. Content Server clients require CS-Direct in order to function. Unless otherwise noted, Content Server clients derive their support from CS-Direct, function with both CS-Direct and CS-Direct Advantage, and are automatically installed. The clients are as follows:

  - **Content Server Desktop (CS-Desktop)** offers content authors the familiar Microsoft Word interface as an alternative to the standard CS interface (which is rendered by CS-Direct). Authors create their content directly in Word documents. However, note that CS-Desktop requires the content in Word documents to be structured, because the content will be parsed to database tables.

    For example, when using CS-Desktop to author content, the user opens a Word document, enters content, and structures the content by tagging it with the same field names as defined in the equivalent content-entry form that Content Server provides. The tagging utility is embedded in the Word interface, and the selection of fields is determined by the CS administrator. When the Word document is

saved, the content in its fields is parsed to fields in the appropriate database table(s). The Word document remains available for editing.

**Note**

CS-Desktop must be manually installed on client machines.

- **Content Server DocLink (CS-DocLink)** supports unstructured content in the flex asset family.

  CS-DocLink provides a drag-and-drop interface for uploading and downloading unstructured types of content—documents, graphics, and other single binary files—that are managed as flex assets. CS-DocLink also presents the hierarchical structure of any flex asset family in the Content Server database as folders and files in the Windows Explorer application.

**Note**

CS-DocLink derives its support from CS-Direct, but functions only with CS-Direct Advantage.

CS-Doc-Link must be manually installed on client machines.

- **InSite Interface** allows users to preview, edit, add, remove, replace, and position content directly on a rendered page. Regular Content Server users can make quick edits in context, while infrequent users can accomplish their work without having to learn the Content Server interface.
- **FCKEditor** is a popular third-party WYSIWYG HTML editor from Frederico Caldeira Knabben. It is the default WYSIWYG editor in the FirstSite II sample site.

  Developers can use FCKEditor to create basic assets whose text-entry fields use FCKEditor as the input mechanism for the field. Developers can also create attribute editors for flex attributes that use FCKEditor as the input medium.
- **Clarkii Online Image Editor (Clarkii OIE)** is a popular third-party image editor from InDis Baltic. Developers can enable Clarkii OIE to allow users to edit images directly in Content Server's interfaces, eliminating the need for an external image editor.
- **eWebEditPro** is a third-party HTML editor from Ektron, Inc. Once configured, eWebEditPro can be used by content providers from within Content Server interfaces.

  Developers can use eWebEditoPro to create basic assets whose text-entry fields use eWebEditPro as the input mechanism for the field. Developers can also create attribute editors for flex attributes that use eWebEditPro as the input medium.

**Note**

In this guide, the term “Content Server” means the Content Server core. The term “Content Server product” means the Content Server core, Satellite Server co-resident, the modules CS-Direct and CS-Direct Advantage, and the Content Server clients.

Additionally, the following features enhance Content Server’s capabilities:

- Satellite Server (remote), a caching application that speeds the performance of your delivery system by serving cached images and Content Server pages from remote servers. You use Satellite Server to reduce the load on the Content Server delivery system and to deliver pages more quickly.

  Note that Satellite Server (remote) is the same application as the co-resident Satellite Server, mentioned earlier in this section. It is offered as a stand-alone product to help you optimize system performance according to the load on the system.
- Engage (available as an optional product), an application that enables your marketing team to divide your site visitors into segments and then target those segments with personalized messages, or promotional, marketing, and informational messages.
- Analytics (available as an optional product), a Content Server plug-in that monitors and statistically analyzes website traffic. Its reporting capability enables you to track visitors’ interactions with published content from the time visitors start browsing your site, up to the time they leave your site.
- Flash content support, which allows users to create, edit, and publish Flash content from Content Server. Developers create Flash template assets which content providers use to assemble Flash content by adding images and text.
- Multilingual asset support, which gives users the ability to assign language designations to assets, and to create translations of assets. Developers also have the option to create site-specific delivery rules for multilingual content that determine which language versions of assets will be shown on the online site, and what to do if a visitor’s request is for a language version in which the content does not yet exist.
- Content Server’s Portal interface, which re-interprets the Content Server Advanced interface as a set of portlets in the content provider’s workspace. One set of portlets is provided for the management of structured content, another set for the management of documents (file-based content).

Each component is summarized below. Readers who are interested in knowing more about the components of the CS product family can find detailed information and references to information in the rest of this chapter, starting on page 36.

## Third-Party Components

The products in the Content Server product family are themselves layered on top of a database management system (DBMS), a web server, and an application server:

- The DBMS stores information about your web site's content.
- Web servers respond to requests for static content by serving HTML pages.
- Application servers are the interface between a DBMS and a web server. Application servers provide fault-tolerance, clustering, and a failover mechanism. The Content Server product makes extensive use of the underlying application server.

The code that you write for your online sites is independent of the DBMS, web servers, and application servers on your content management systems. With only a few exceptions, the code that you write for one configuration will continue to work when moved to a new configuration.

## J2EE Compliance

Java 2 Platform, Enterprise Edition (J2EE) is an industry standard for developing multi-tier enterprise applications. J2EE is a framework for simplifying enterprise applications by basing them on standardized, modular components; by providing a complete set of services to those components; and by handling many details of application behavior automatically, without complex programming.

All of the applications in the Content Server product family conform 100% to the J2EE standard.

## Content Server Systems

When you are using Content Server for your content management needs, you and the others on your team work with up to four different systems:

- **Development** system, where developers and designers plan and create the online site.

  All of the CS products that you have purchased are installed on this system (including Engage, if you are using Engage).

- **Management** system, where content providers such as writers, editors, reviewers, graphic artists, product managers, and marketers develop the content that is delivered to visitors of the online site. Revision tracking and workflow features track changes to assets (content), monitoring them until they are approved to be published to the delivery system.

  Only Content Server and Engage (if it is being used) are installed on this system.

- **Delivery** system, where the content you are making available or the products that you are selling are served to your visitors or customers.

  If you are delivering your content dynamically, all of the Content Server products that you purchased are installed on this system. If you are delivering your content statically—that is, if you are serving static HTML pages—your delivery system is a web server only, and you do not need to install any of the Content Server products on that system.

- **Testing** system, where you or your QA engineers test the performance of both the management system and the delivery system. If a dedicated testing system is not available, testing can be done on the development system.

As a developer, you spend the majority of your time working on the development system. When the asset types that you develop and the site that you have designed are ready, you migrate your work from the development system to the management system. As assets are created, modified, and approved by the content providers on the management system, they are published from the management system to the delivery system.

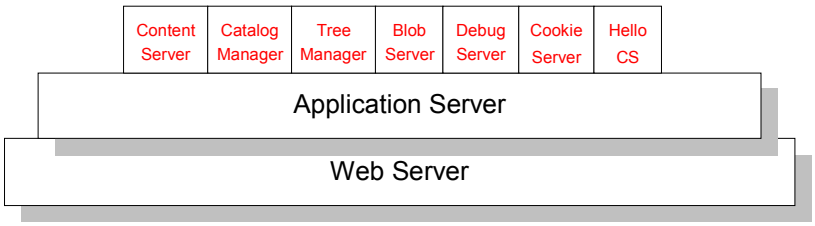
# The Content Server Core

The Content Server core is the operating system that powers the entire Content Server product family. Although the CS modules make it easier for you to design the data structure of the data that you want to serve from your online site, and to design and assemble the online site, it is the Content Server core that actually serves that data.

In addition to being an operating system that runs the product family and serves your content, the Content Server core is a toolset. It provides Java methods and utilities that you can use for designing your online site, for developing your own content management applications, and for customizing the Content Server modules/products.

## Servlets and Java APIs

The Content Server operating system consists of several servlets that run on top of an application server. Each servlet is invoked when necessary to perform a discrete set of tasks. Each servlet has a corresponding Java API with Java methods and custom XML and JSP tags that you use to invoke the functions that you need to use. Servlets are shown in the following figure:




The Content Server servlets are as follows:

- **ContentServer** – Generates and serves pages dynamically. This servlet provides disk caching, session management, event management, searching, and personalization services.

  **Note**

  The ContentServer servlet (one word) is distinct from Content Server (two words), which refers to the application.

- **CatalogManager** – Provides most of the database management for the Content Server database, including revision tracking, security, resultset caching, and publishing services.
- **TreeManager** – Manages the tree tables, which store hierarchical information about other tables in the Content Server database.
- **BlobServer** – Locates and serves binary large objects (blobs). Blobs are not processed in any way—they are served as is, as they are stored.
- **DebugServer** – Provides tools that help you debug your XML code.
- **CookieServer** – Serves cookies for Content Server pages, whether those pages are delivered by the ContentServer servlet or by the Satellite Server application.

- **HelloCS** – Displays version information about the Content Server software installed on your system.

In general, you do not need to know which servlet performs which service or task. You simply invoke the appropriate Java method or XML or JSP tag and let the Content Server core application determine which servlet to call. The exception to this rule is when you write code that references a servlet URL; that is, when you include a link to a blob or to another page on a Content Server page. Because the ContentServer servlet and the BlobServer servlet reside at different URLs, you must include the URL of the appropriate servlet in your `<A HREF>` tags.

For information about the coding links to blobs and pages, see Chapter 4, "Programming with Content Server" and Chapter 27, "Coding Elements for Templates and CSElements."

## Publishing APIs

You make content available to the visitors of your online site by moving it from your management system to your delivery system. This process is called **publishing**.

The Content Server core application provides two publishing APIs: Export and Mirror. The Export API supports static pages by rendering your Content Server pages into static HTML files. The Mirror API supports dynamic pages by copying publishable data from tables in the Content Server database on your management system to the corresponding tables in the Content Server database on your delivery system.

Content Server's publishing APIs are the foundation of the publishing functionality that CS-Direct delivers, in particular the two publishing options called "Export to Disk" and "Mirror to Server."

**Note**

In this guide, "Export to Disk" is also called "static publishing."

"Mirror to Server" is called "dynamic publishing."

- For information about the publishing APIs, see the *Content Server Javadoc*, *Content Server Administrator's Guide,* and the *Content Server Tag Reference*.
- For information about the publishing features delivered with the Content Server products, see the publishing chapter in the *Content Server Administrator's Guide*.

## Content Server Utilities

As previously mentioned, Content Server provides GUI-based developers' utilities for managing the Content Server database in a way that facilitates the development and maintenance of web sites. The utilities are CS-Explorer, CatalogMover, XMLPost, BulkLoader, and Property Editor.

The utilities are used together with the Content Server browser-based interface. Chapter 8, "Content Server Tools and Utilities" provides brief descriptions of the utilities, and tells you how to start them.

Other chapters provide in-depth information regarding the usage of the utilities in the context of basic assets, flex assets, non-asset tables, elements, and pages:

- For additional information about CS-Explorer usage, see:
  - Chapter 17, “Flex Filters,” in the section “Defining a Flex Filter Class and Creating a Flex Filter Asset” on page 337.
  - Chapter 23, “Creating Template, CSElement, and SiteEntry Assets”
  - Chapter 27, “Coding Elements for Templates and CSElements”
- For additional information about CatalogMover usage, see Chapter 23, “Creating Template, CSElement, and SiteEntry Assets” (pages 497 and 500).
- For additional information about XMLPost usage, see:
  - Chapter 11, “Data Design: The Asset Models”
  - Chapter 15, “Designing Basic Asset Types”
  - Chapter 16, “Designing Flex Asset Types”
  - Chapter 20, “Importing Assets of Any Type”
  - Chapter 21, “Importing Flex Assets” (entire chapter)
  - Chapter 22, “Importing Flex Assets with the BulkLoader Utility”
  - Chapter 25, “Creating Collection, Query, Stylesheet, and Page Assets”
- For additional information about BulkLoader usage, see:
  - Various sections of Chapter 21, “Importing Flex Assets”
  - Chapter 22, “Importing Flex Assets with the BulkLoader Utility”
- Information about various properties in the Property Editor is presented throughout this guide as necessary. All properties are described in detail in the *Content Server Property Files Reference*.

## Search Engine Interfaces

The Content Server core does not provide a search engine but it does provide an interface to the Lucene third-party search engine. (The Lucene search engine ships fully integrated with Content Server.) For information about using the search engine on either your management system or your delivery system, see the *Content Server Administrator’s Guide*.

## Page-Generation Components

While the Content Server core supports both static and dynamic pages, dynamic page generation is its main function, and is the subject of this section.

**Note**

In this guide, “static pages” are also called “HTML pages.” “Dynamic pages” are called “Content Server pages” to make it clear that they are dynamic pages generated by the Content Server core application.

A dynamic CS page differs from a typical HTML page, as shown in the following table:

| Static Page (HTML Page) | Dynamic Page (Content Server Page) |
|---|---|
| Single disk file, served via a web server.<br><br>Because an HTML page is not dynamic, each request can display only one and the same page. | Generated upon request.<br><br>Content Server makes dynamic content decisions based on the request. Different requests for the same page typically result in the generation of different pages. |
| One-to-one association between the HTML page and the page the visitor sees in the web browser. | The web page that the visitor sees can be composed of multiple components called “pagelets,” created from within Content Server. |
| No separation of format and content.<br><br>As a result, it is difficult to modify format and content independently of each other. | Separation of format and content.<br><br>As a result, format and content can be modified and maintained independently of the other. |

## Page Formatting Code: Element Files

In very simplistic terms, Content Server’s main function is to separate format from content. By separating the two, Content Server enables you to reuse the same bits of formatting code for many pieces of content. For example, if you want to change the format of article assets, you rewrite the code in one place, rather than having to rewrite code for every article in your system.

Your formatting code (Java, XML, JSP, HTML, JavaScript, and so on) is stored in files called **elements**. The code extracts the content from the database and formats the content. Because content is formatted only when a page is requested, you have the opportunity to design pages that will be constructed on-the-fly, according to the identity of the visitor requesting them.

## Pagelets

A page in the Content Server environment is the result of an HTTP request displayed in a browser. Content Server creates a page by compiling **pagelets** (page components, such as headers and footers), into the final rendered page. The page is the output (HTML, XML, WAP, and so on) that Content Server generates when it parses your JSP or XML elements and blobs.

## Page Rendering: The ElementCatalog and SiteCatalog Tables

Element files are stored in the `ElementCatalog` table in the Content Server database. The names of your pages and pagelets (parts of the pages) are stored in the `SiteCatalog` table. That is, the `SiteCatalog` table stores the entries for all the legal page names for your online site.

Each row in the `SiteCatalog` table is a page entry. Each page entry points to an element in the `ElementCatalog` table. The element being pointed to by a page entry is called the **root element** of the page entry.

Content Server renders your content into an online page by executing `SiteCatalog` page entries. Here's how it works:

1. A visitor enters a URL to your online site in a browser.
2. The web server that processes the HTTP request maps that URL to a Content Server URL. For example, this is a Content Server URL:

   ```
   http://www.BurlingtonFinancial.com/servlet/
   ContentServer?pagename=BurlingtonFinancial/Home
   ```

   The text at the end of a Content Server URL is called the pagename. In this example, the pagename is `BurlingtonFinancial/Home`.
3. Content Server looks up the page name in the `SiteCatalog` table, determines its root element, locates that element in the `ElementCatalog` table, and then invokes that element.
4. The element is executed. If there are calls to other elements from within the root element, those elements are executed in turn.
5. The results—images, articles, and so on, including any HTML tags—are rendered into HTML code and returned to the visitor's browser.

The result is a page that is dynamically rendered on demand.

## Page Design Considerations

Because Content Server separates format from content, it encourages you to design pages in a modular fashion—by assembling the pages from pagelets (as mentioned above, pagelets can be headers and footers. They can also be bylines, sidebars, and so on). A modular page design provides the advantage of reusability—one set of pagelets can be reused to construct multiple pages, but can be maintained in one place.

When designing pages, consider also system performance. For best performance, take into account your page caching strategy. For information about page caching, see "Database Management Functions" on page 41, and Chapter 5, "Page Design and Caching." See also "Satellite Server" on page 43.

## Database Management Functions

The Content Server product is database-driven; most information in Content Server and its product family is represented as a row in a database table. For example: pagenames are stored as rows in the `SiteCatalog` table and elements are stored as rows in the `ElementCatalog` table. Therefore, managing assets—content, pagelets, pages, and elements—involves managing the database tables where they are stored.

**Note**

In older versions of Content Server, database tables were called catalogs. As a result, some tables still have the word "catalog" as part of their names.

A complete listing of supported DBMS systems is available in the *Supported Platform Document (SPD)*, at the following URL:

`http://e-docs.fatwire.com`

The Content Server core, which makes extensive use of the underlying database management system, supports five types of database tables: object, content, tree, foreign, and system tables (all described in Chapter 12, "The Content Server Database"). The Content Server core also keeps track of all the tables on a given system by keeping a record of them in the `SystemInfo` table, which is a table of tables.

Additionally, the Content Server core provides developers with several database management functions and tools:

- Configurable page caching (for example, you can set page expiry time)
- Configurable resultset caching
- Revision tracking that can be enabled or disabled
- System security tools, such as Access Control Lists (ACLs), to manage users' access to CS systems and control visitors' access to information on the online site

## Page Caching

Page caching plays a significant role in system performance. If an element is not changed and it will generate the same page each time it is invoked, why make Content Server process the element each time it is called? If the generated page is cached, it can be served much faster than it can if it must first be generated.

The Content Server core alone (independently of Satellite Server) can separately cache each page or pagelet that is identified by a page entry in the `SiteCatalog` table. You can mark the expiration date of any pagelet in the cache by specifying a value for that page entry in that table.

Page caching is made especially effective by the addition of Satellite Server. Installing a Satellite Server application amounts to installing page caches on the servers that host Satellite Server, thereby extending the Content Server page cache.

- For information about page caching, see Chapter 5, "Page Design and Caching."
- For information about Satellite Server, see "Satellite Server" on page 43.

### Resultset Caching

Resultset caching is another feature that can greatly enhance system performance. When the Content Server database is queried by any mechanism, the Content Server application can cache the resultset that it returns. The Content Server application keeps track of every table in the database; whenever a table is modified, it flushes all the resultsets that were cached for that table.

You configure resultset caching through properties in the `futuretense.ini` file.

For more information about resultset caching, see Chapter 14, "Resultset Caching and Queries."

### Revision Tracking

Content Server core provides revision tracking functionality that prevents a row in a table from being edited by more than one user at a time. When you enable revision tracking for a table, Content Server maintains multiple versions of each row in a table.

### Security and User Management

Security features, such as Access Control Lists (ACLs), in the Content Server core allow you to limit access to:

- Individual database tables
- Individual Content Server pages

In other words, not only can you control which users of your content management system can add or change data in the Content Server database, you can control which visitors are allowed to see which pages in your online site.

- For information about user management on the management system, see the *Content Server Administrator's Guide*.
- For information about user management on the delivery system, see Chapter 31, "User Management on the Delivery System."

## Sessions and Cookies

Content Server automatically creates a session for a visitor when he or she visits your online site for the first time. You can store information about that visitor in session variables by using the tags and methods in the Content Server core. Subsequent elements can then access those variables and respond conditionally to them.

Session variables, however, are volatile. They last only as long as the session lasts, that is, until one of the following events occurs:

- The visitor closes his or her browser.
- The session times out after a period of inactivity. You control session timeouts through a property in the `futuretense.ini` file.
- The application server is restarted (except in a cluster).
- The session is disabled in some other way.

To store information on a more permanent basis, you would use cookies. You can code your elements to write cookies that store information about your visitors to their browsers. Then, you can use the stored information to customize pages and display the appropriate version of a page to the appropriate visitor when he or she returns to your online site.

For more information about sessions and cookies, see Chapter 9, "Sessions and Cookies."

## Event Management Features

Many users are familiar with `cron`, a daemon that runs certain programs at a specified time and place. Content Server provides a similar feature called “event management.” This feature enables you to identify elements that should be run at a certain date and time. For example, you might schedule a Content Server event to occur on a nightly basis to delete voided assets from the Content Server database.

# Satellite Server

Your installation of Content Server includes Satellite Server, a program that caches the pages and pagelets that you create. By default, Satellite Server is installed on the same machine where Content Server is installed. This co-resident Satellite Server works in combination with Content Server to provide double-buffered page caching.

You can further improve your system’s performance by installing Satellite Server remotely so it can cache pages and pagelets closer to their intended audience. Remote Satellite Server hosts are fast, inexpensive caches of Content Server pages. They reduce the load on the Content Server host, dramatically increase the speed of page delivery to your site visitors, and provide a simple and inexpensive way to scale your Content Server system.

In order to use Satellite Server to cache your pages and pagelets, you include Satellite Server XML or JSP tags in your page elements. These tags tell Satellite Server which parts of which pages should be held in its cache. After you have coded your web site with Satellite tags, you then call your pages with Satellite URLs.

## Handling the HTTP Requests

When the load balancer routes an HTTP request for a page to Satellite Server, the following chain of events occurs:

1. Satellite Server checks its cache.
2. If the page is in the cache, Satellite Server serves it to the visitor’s browser.
3. If the page is not in the Satellite Server cache, it routes the request to Content Server.
4. If Content Server has the page in its cache, it returns the cached page to Satellite Server. If the page is not in the Content Server cache, Content Server renders the page, caches a copy, and sends the page to Satellite Server.
5. Satellite Server caches the page and serves it to the visitor’s browser.

Each Satellite Server application is independent of every other Satellite Server application. An individual Satellite Server application has the following characteristics:

- It maintains its own cache.
- It cannot mirror its cache to a cache on another server maintained by Satellite Server.
- It cannot request pages or pagelets from another Satellite Server application. It can request pages or pagelets from only the Content Server core.

## Satellite Server Servlets and APIs

Satellite Server is made up of several servlets: one that caches and serves pages, and two that manage the cache:

- **Satellite** – Caches pages at the pagelet level. The Satellite XML or JSP tags in your elements indicate which pagelets should be cached, and they control various Satellite Server settings.
- **Inventory** – Enables you to examine the Satellite Server cache so you can obtain the information you need to manually flush individual pages or pagelets from the cache when necessary.
- **FlushServer** – Handles all types of cache-flushing. FlushServer can either flush the entire cache, or can flush individual items from the cache.

For information about coding pages with the Satellite Server tags and page caching in general, see Chapter 5, "Page Design and Caching."

# CS-Direct

CS-Direct is the base module, built with the Content Server Java APIs. CS-Direct provides the Advanced, Dash, and InSite interfaces to the Content Server database, enabling you to easily organize, categorize, manipulate, and maintain your content. Content is stored in the CS database as objects called **assets**.

The CS-Direct application also provides direct access to all of Content Server's core features, thereby adding another layer to the content management framework. CS-Direct introduces the following concepts and features to the Content Server product family:

- The basic asset model to support flat data structures
- The Advanced interface for developers (and administrators), as well as content providers and business users
- The Dash, InSite, and Portal interfaces for content providers and business users
- Client interface options: CS-Desktop and CS-DocLink
- The content management site and the site plan
- The sample sites FirstSite II, Burlington Financial, and Hello Asset World
- Extension of the Content Server rendering model through Template, CSElement, and SiteEntry assets
- Custom XML and JSP tags
- Enhanced publishing functionality: publishing methods and the approval subsystem
- Enhanced revision tracking functionality
- Workflow
- Additional search engine features

You use CS-Direct to design the data that you want to store in the Content Server database, to create and manage the content in the system, and to design the online site that you present on your delivery system.

Because nearly everything in Content Server and the Content Server product family is represented as a row in a database table, all of the CS-Direct features and functions are accounted for in the Content Server database.

## Basic Asset Model

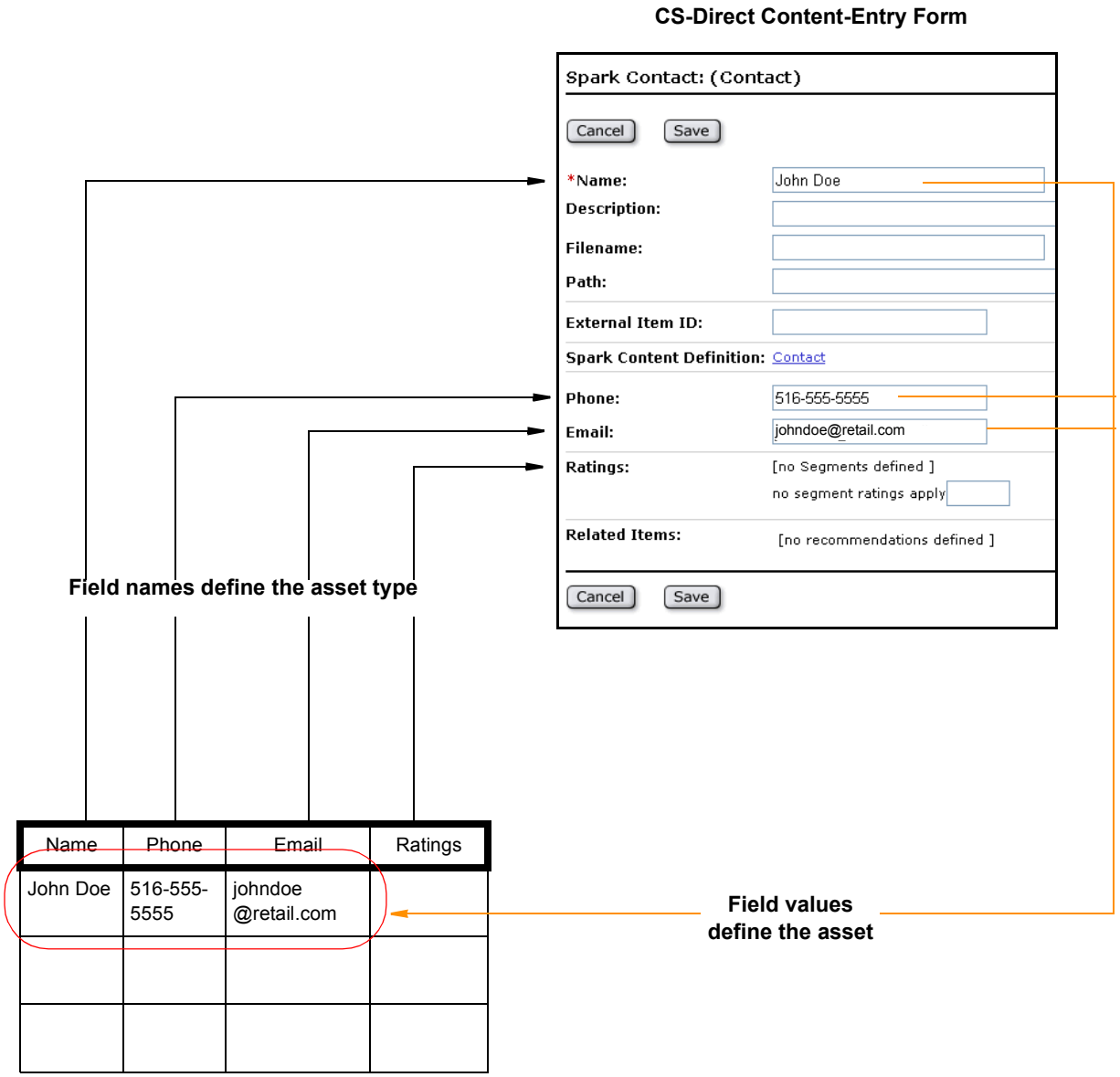
CS-Direct provides a framework for your content, in which that content is stored as objects called **assets** in the Content Server database. The framework is called the **basic asset model**, which allows one primary storage table in the database for each type of asset. When assets are created from an **asset type**, they can be associated with other assets, but they cannot inherit each other's properties. Thus, only flat data structures are possible with the basic asset model.

CS-Direct also provides interfaces (Advanced and Dash) to the database tables where the assets are stored. The interfaces consist of forms for creating and managing assets, as well as forms for creating and managing asset types. The relationship of the database tables to the content management forms (rendered by the Advanced interface) is shown in Figure 1, "Asset Types: Content-Entry Forms and Database Tables" on page 46.

Note that while the actual process of creating assets and asset types depends on the module or add-on that is being used, the concept of assets and asset types is not specific to CS-Direct. Rather, it is introduced by CS-Direct, but used throughout the Content Server product family.

**Figure 1:** Asset Types: Content-Entry Forms and Database Tables

When rendering the content-entry form below, CS-Direct displays the names of columns in the database table as field names in the form. Field names (specified by developers) define the asset type; field values (entered by content providers) define the asset.




**Database Table for Asset Type "Contact"**

Regardless of the module or add-on, assets are created from **asset types**, which are of two varieties:

- Asset types can be content types. For example, developers might create asset types such as articles and images; from these asset types content providers would create assets (instances of the asset types) such as a news article, a financial article, or a sports image.

  Assets, in general, are objects that can be created, edited, inspected, deleted, translated, duplicated, placed into workflow, tracked through revision tracking, searched for, previewed, and published, all of which are functions in the CS-Direct application.

  During the process of designing your online site, you and the others on your team examine your design and determine which parts should be assets. Data that should be treated as an asset must not be embedded into code.

- Asset types can be types of logic. For example, the following core asset types are used as site design features in CS-Direct and are therefore classified as logic asset types: page asset, collection, query asset, Template asset, CSElement, and SiteEntry. Another example of a logic asset type is “stylesheet,” delivered with the Burlington Financial sample site.

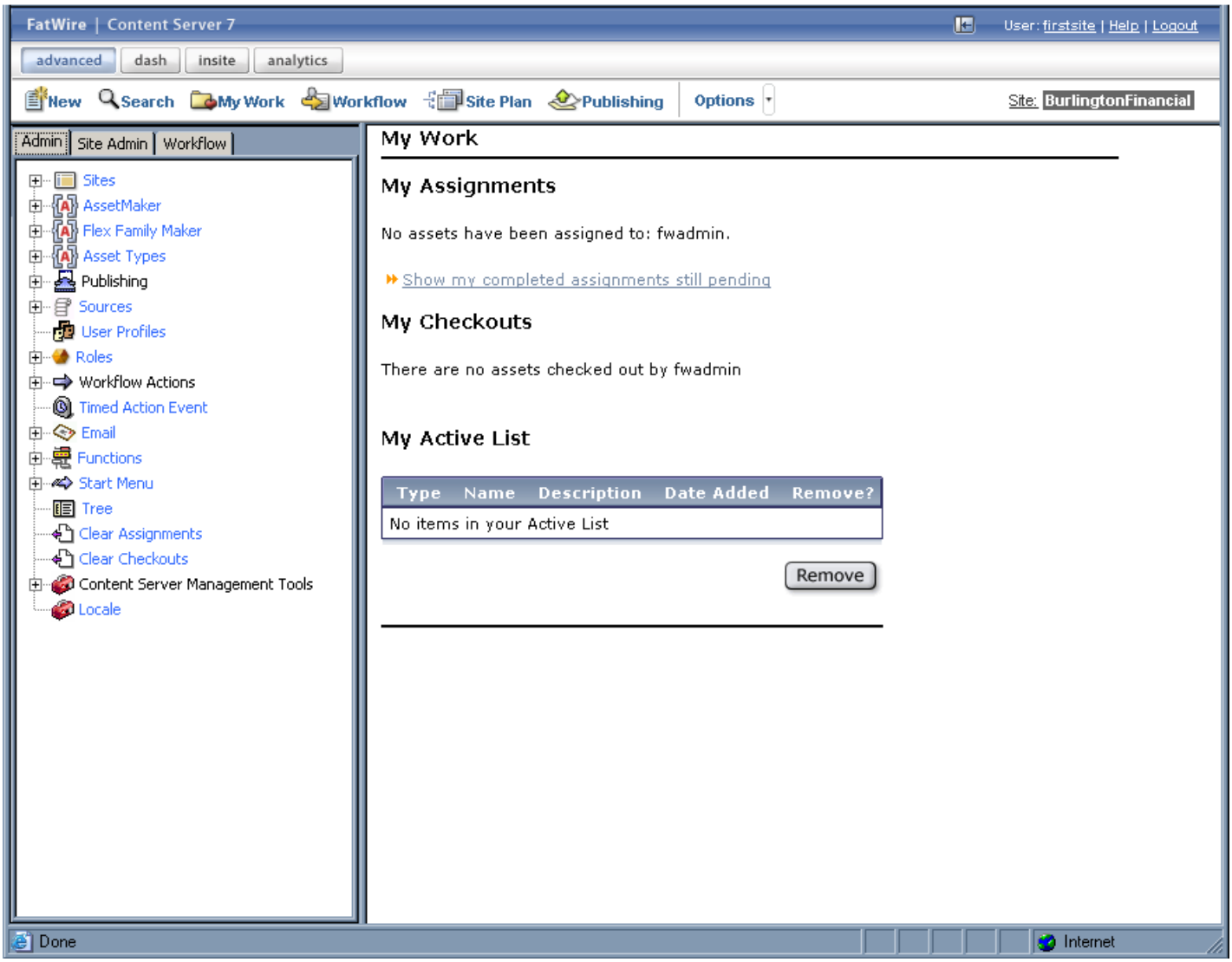
  Like content assets, logic assets use the asset data model, so that you can use CS-Direct content management features (workflow, revision tracking, access control, and so on) to maintain them.

For more information about basic assets and asset types, see:

- Chapter 11, “Data Design: The Asset Models”
- Chapter 15, “Designing Basic Asset Types”

## The Advanced Interface

CS-Direct renders the Advanced interface (shown below) designed for developers, administrators, as well as content providers.




The tree panel on the left contains all the content management elements that developers, administrators, and content providers need to work with. The workspace area on the right is where all the tasks and operations are performed.

**Note**

All procedures in this guide are based on the Advanced interface.

The interface supports code-based operations, and enables you to graphically complete the creation of basic asset types. For example, to create a basic asset type, you would:

1. Write an XML file (called “asset descriptor” files) to define the basic asset type.

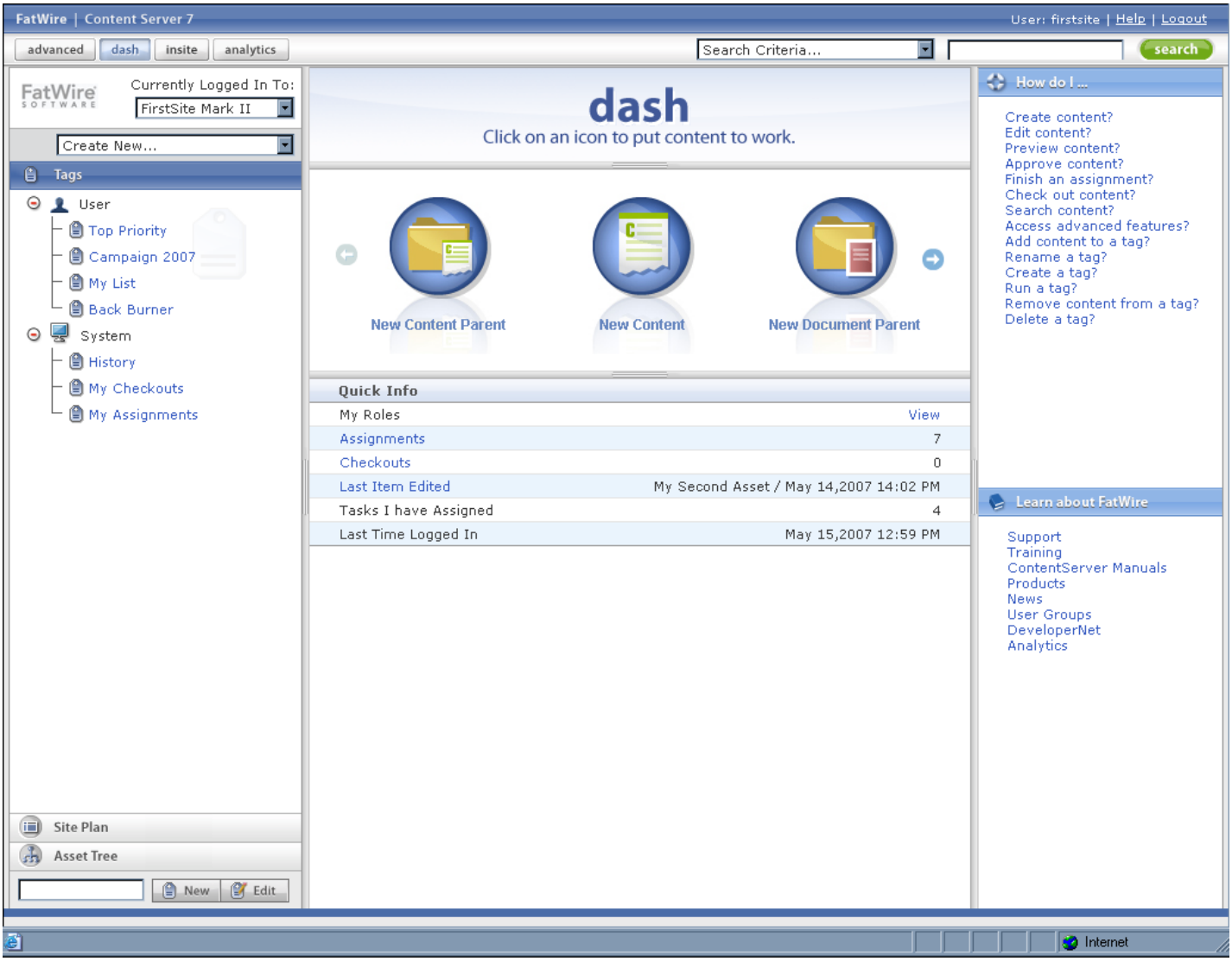
2. Upload the file to CS-Direct.
3. Use the CS-Direct interface to invoke the AssetMaker utility. One of the functions of the interface (AssetMaker) is to read the asset descriptor file and, from it, create a storage table for the asset type. Other functions in the interface allow you to configure the asset type (for example, name its authorized users).

The same interface is used by administrators to create content management sites, manage system users, control their permissions to content, establish workflow processes, and configure Content Server features (such as CS-Desktop).

In the same interface are content management functions, used mostly by the content providers. There are also forms for entering, editing, inspecting, and otherwise managing basic assets in the database. An example of such a form and its relationship to the database is shown in Figure 1, “Asset Types: Content-Entry Forms and Database Tables” on page 46. (Note that the relationship in Figure 1 does not strictly apply to the flex asset model, because the flex asset model creates multiple database tables per asset type.)

## The Dash Interface

CS-Direct renders the Dash interface (shown below) designed specifically for content providers and business users. The Dash interface provides ease of use and quick access to most of Content Server's content management functions.




The Dash interface supports most of the Content Server features available in the Advanced interface. For a feature comparison list, see the overview chapter in the *Content Server Dash Interface User's Guide*.

## Content Server Clients (Interface Options)

CS-Direct also supports clients (optional interfaces) such as MS Word and applications that offer functionality similar to Windows Explorer and Windows Desktop. For more information, see "Content Server Clients (Interface Options)" on page 58.

## Sites and the Site Plan

CS-Direct introduces the concept of a “content management site” to the Content Server product family. A content management site is an object that you use as an organizational construct for an actual online site and as an access control tool. You use sites to control access to assets and to facilitate the designing of your online site.

When you log in to Content Server, you are logging in to a content management site. If you have access to more than one site, the first decision that you make after logging in is which site to work on.

**Note**

For simplicity, this guide often refers to “content management site” as “site.”

A content management site represents a real, online site. However, it can represent that online site in any number of ways, depending on what makes sense for your situation. For example, you could create separate content management sites for separate sections of your online site because the teams who provide content for each section work completely separately from each other and only members of that team should have access to that section (content management site). Or, you could create a content management site that represents an entire online site, as does the Burlington Financial sample site.

When you install CS-Direct, the **Site Plan** tab and the rest of the tree appear in the Content Server interface. The **Site Plan** tab displays a representation of the site design for the content management site that you are currently logged in to.

For information about content management sites and the **Site Plan** tab, see Chapter 2, “Overview of Sites.” See also the introductory chapter in the *Content Server Administrator’s Guide*.

## Sample Sites

CS-Direct provides the sites described in this section.

### The FirstSite II Sample Site

FirstSite II is FatWire's flagship reference implementation. Utilizing our best practices in Content Server site development, FirstSite II provides an excellent starting point for developers. As with all FatWire sample sites, FirstSite II code can be reused and modified to suit your organization's business needs. This guide uses FirstSite II examples to illustrate best coding practices. For information on the FirstSite II sample site, see the guide *FirstSite II: Use, Reuse, and Replication*.

### The Burlington Financial Sample Site

CS-Direct provides a fully functional sample site named Burlington Financial. The site is used in this guide as the source of examples that illustrate the basic asset model, CS-Direct functionality, and coding practices. You can examine the examples both in this book and online in the context of the actual site.

The Burlington Financial site contains sample asset types, elements, `SiteCatalog` entries, a workflow process, and so on. For information about the Burlington Financial sample site, see Chapter 33, “The Burlington Financial Sample Site.”

### The HelloAssetWorld Sample Site

In addition to the Burlington Financial sample site, CS-Direct delivers a sample site called "HelloAssetWorld." The site provides a simple introduction to creating a Content Server web site.

The templates that compose HelloAssetWorld are described in Chapter 32, "The HelloAssetWorld Sample Site." Further information about the site's configuration and users is available in the *Content Server Administrator's Guide* and the *Content Server Advanced Interface User's Guide*.

## Template, CSElement, and SiteEntry Assets

CS-Direct provides an additional layer in the Content Server page rendering process: the Template, CSElement, and SiteEntry asset types.

Template, CSElement, and SiteEntry asset types provide the elements and pagelets that build your online sites. They are asset representations of elements and page names, the components that Content Server uses to generate pages. Because they are assets, elements and page names can be managed with workflow and revision tracking.

For information about template, CSElement, and SiteEntry assets, see Chapter 23, "Creating Template, CSElement, and SiteEntry Assets."

## Custom CS-Direct XML and JSP Tags

CS-Direct delivers several new tag families (both XML and JSP versions) that you use to code your elements. The tag families enable you to identify, extract, and then display assets on your online site.

- For information about coding pages that display assets that use the basic data model, see Chapter 27, "Coding Elements for Templates and CSElements."
- For information about all of the CS-Direct custom tags, see the *Content Server Tag Reference*.

## Approval and Publishing

As mentioned previously, a Content Server page is a set components that have been assembled into a viewable, final output. Creating that output is called **rendering**. Making either that output or the content that is to be rendered available to the visitors of your online site is called **publishing**

You publish by copying your content from your management system to the delivery system. CS-Direct delivers two publishing methods that are built from the Content Server publishing APIs. These publishing methods interact with the CS-Direct approval system, an underlying system that determines which assets have been approved.

When assets are ready to be published, someone marks them as approved. Then, when the publishing process is ready to start, it invokes the approval system which compiles a list of all the approved assets and examines all the dependencies for those assets. If an asset is approved but an asset that it is linked to is not approved, the approved asset is not published until the linked asset is also approved.

The CS-Direct publishing and approval systems track and verify all the asset dependencies in order to maintain the integrity of the content on your delivery system. The publishing

and approval systems ensure that the assets which you have determined to be ready for publishing are the only assets that get published.

The CS-Direct publishing methods are as follows:

- **Mirror to Server** is the dynamic publishing method. It is built with the Content Server Mirror API to copy approved assets from the Content Server database on one system to the Content Server database on another system.
- **Export to Disk** is the static publishing method. It renders your approved assets into static HTML files, using the template elements assigned to them to format them. An administrator or automated process then copies those files to your delivery system using FTP or another file transfer method.

For information about configuring publishing, see the *Content Server Administrator's Guide*.

For information about coding elements so that they log dependencies appropriately and how CS-Direct calculates approval dependencies, see Chapter 27, "Coding Elements for Templates and CSElements."

For information about how you approve assets, see one of the Content Server user's guides.

## Revision Tracking

When you are using CS-Direct, the Content Server revision tracking feature is extended and implemented for asset types in the Content Server interface. There are additional administrative forms and the asset forms are modified to include revision tracking functionality when it is enabled for that asset type. You specify which asset types must have their assets tracked. CS-Direct enables the revision tracking feature for the appropriate tables without your having to do so directly through the Content Server core. Content providers can then check out assets from the database and check them back in as they work.

For information about revision tracking, see the *Content Server Administrator's Guide*.

## Workflow

CS-Direct introduces the workflow feature to the Content Server product family. Workflow is the movement of content from one person to another in a predictable, systematic way.

For example, perhaps all articles must be reviewed by both an editor and by someone from your legal department before they can be approved (and then published). You can use the workflow feature to ensure that an article is assigned to the appropriate person at each stage of its life cycle, and to restrict unauthorized users from accessing the article at each stage of the workflow.

- For information about creating workflow processes, see the *Content Server Administrator's Guide*.
- For information about using the workflow feature to obtain and finish work assignments, see the *Content Server Advanced Interface User's Guide*.

## Searching and Search Engines

CS-Direct comes with a search function that helps users find the assets they want to work with. The Advanced interface uses an SQL-based database search mechanism, while the Dash interface is fully integrated with the Lucene search engine.

To enable searches for assets that you want to display in your online site on your delivery system, you use a combination of SQL queries, query assets, and collection assets. In certain cases, you might also want to use a search engine on your delivery system so that you can implement an index-based search feature for online site.

For information about search engines, see the *Content Server Administrator's Guide*.

# CS-Direct Advantage

The CS-Direct Advantage module is built on top of CS-Direct. Independently of the basic asset model, it offers a more advanced asset model, and facilitates the development of online businesses by providing support for e-commerce applications. CS-Direct Advantage introduces the following concepts and features:

- Flex Asset Model
- GE Lighting Sample Site
- Assetsets and Searchstates: Searching the Online Site
- Shopping Carts and Commerce Context
- Custom CS-Direct Advantage XML and JSP tags

## Flex Asset Model

Whereas the basic asset model is used for creating flat data structures with small amounts of data, the flex asset model is used for creating hierarchical data structures and structures with large amounts of data (historically it has been used for creating large, online catalogs of products). Even if you are designing a flat data structure, you might want to consider basing it upon the flex asset model for the following reason: The flex asset model fully supports changes to database schema, making flat data structures easily convertible to hierarchical structures.

Note that the flex and basic asset models are not interchangeable. So, if you elect to create a flat data structure using the basic asset model and then decide to convert the data to a hierarchical structure, be prepared to re-create the data in the context of the flex asset model.

The flex asset data model differs from the basic model in other ways. Basic assets and all of their attribute values are stored in one, primary storage table such that basic assets of the same type have the exact same attributes (properties). Flex assets, on the other hand, have several storage tables. In addition, attributes are stored in a way that supports inheritance; that is, attribute values are passed on from parent assets to their child assets. If different parents are chosen for assets of a given type, those assets will vary widely. Thus, the flex data model supports variable content and lends variability when it is required.

Furthermore, you do not create individual flex asset types as you do basic asset types; instead, you create a flex family of asset types. There are five *required* members in a flex

family. You must create all of them. You can then use the members selectively to create either flat or hierarchical data structures, as shown in Table 1, "Flex Family Members."

**Table 1:** Flex Family Members

| Flex Family Member | Flat Data Structure | Hierarchical Data Structure |
|---|---|---|
| flex attribute asset type | ✓ | ✓ |
| flex parent definition asset type | | ✓ |
| flex parent asset type | | ✓ |
| flex definition asset type | ✓ | ✓ |
| flex asset type | ✓ | ✓ |
| flex filter asset type (optional) | ✓ | ✓ |

Notice that hierarchical organization is made possible by the inclusion of the flex parent definition asset type and the flex parent asset type. The sixth member of a flex family, the flex filter type, is optional and can be used in both flat and hierarchical structures.

Each asset type is expressed as a data-entry form that developers use to create instances of that type. The flex attributes asset type is used to create flex attributes, the flex parent definition type is used to create flex parent definitions, and so on.

The key member of a family is the **flex asset type**, from which content contributors create flex assets, units of content that are meant to be extracted from the database and displayed to visitors of the online site (delivery system). In a hierarchical structure, all the other members in a flex family contribute to the flex asset as explained in this guide. For readers who prefer to learn by example, this guide provides a tutorial on flex assets, explaining the function and effects of each asset type as it is being created.

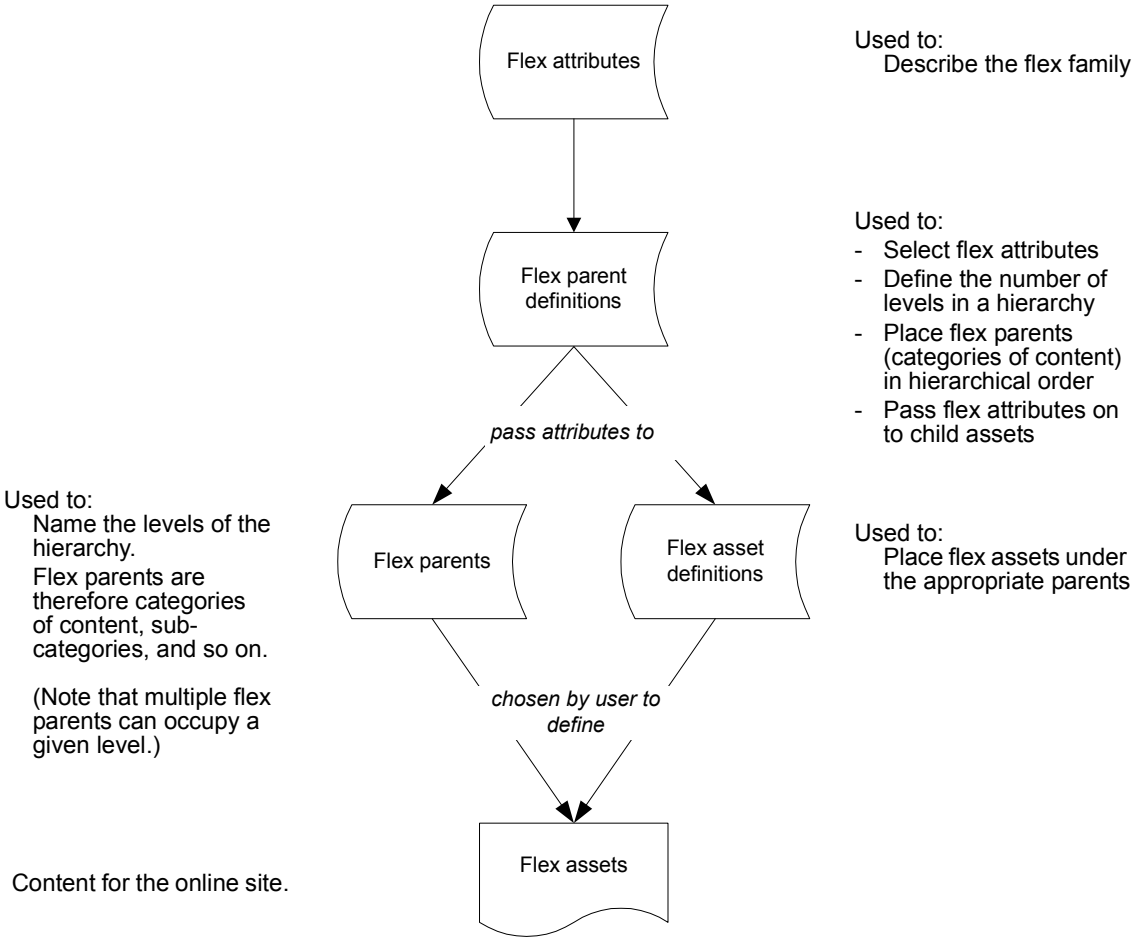
## General Information About Flex Families

The basic relationship among flex assets is illustrated in Figure 2, "Attribute Inheritance Tree." Arrows show how attributes are passed down to assets: The attributes are first used to create the flex parent definition(s). The flex parent definitions are used to establish the levels of a flex family's hierarchy and to place parent assets in hierarchical order. Flex definitions are used to place flex assets under the appropriate flex parents, allowing the child assets to inherit the attributes that were given to their parents by the flex parent definitions.

**Note**

If you plan to use the flex asset model to create a flat data structure, you can omit creating flex parent definitions and flex parents. However, you must create one or more flex attributes and one or more flex definitions. This allows you to assign the flex attributes directly to the flex definition assets, and therefore, to assets in the family.

**Figure 2:** Attribute Inheritance Tree



Following are some general characteristics of the flex asset model:

- Flex parents and flex assets are described by the flex attributes that you select for them.
- The attributes that characterize flex parents and flex assets are themselves assets. This means that attributes can be passed through workflow, edited, monitored by revision tracking, and subjected to all other content management operations.
- In hierarchical data structures, flex assets inherit attribute values from their parents. The flex definition asset types combined with the inheritance of attributes enables you to set up group hierarchies and implement some sort of taxonomy with your data.
- If you ever need to add attributes to your asset types in the future (a common occurrence with products), you just create the new attribute and assign it to the appropriate definitions. In contrast, with the basic asset model, you cannot add more attributes after you have created the asset type.
- By using the flex definition asset type, you can set up multiple "templates" for the same flex asset type.

This asset model supports assets that have many, many attributes, which means that you can support large sets of data.

For more information about the flex asset model, see:

- Chapter 11, "Data Design: The Asset Models"
- Chapter 16, "Designing Flex Asset Types"

## Assetsets and Searchstates: Searching the Online Site

The flex asset data model delivers a mechanism, called the "searchstate," to use on the pages that extract and display flex assets.

A **searchstate** is a set of search constraints that are applied to a list or set of flex assets, which is an **assetset**. A constraint can be a filter (restriction) that is based on either the value of an attribute or on another searchstate (called a nested searchstate).

A searchstate can search either the attribute tables in the database or the attribute indexes created by a search engine.

If you are using the flex asset data model, searchstates provide two advantages:

- You can mix database (parametric) and rich-text (full-text through an index) searches to compile a single assetset.
- There is no need to write SQL queries. If a new version of the CS-Direct Advantage application introduces a schema change, your searchstate code does not need to change.

For more information about assetsets and searchstates, see Chapter 27, "Coding Elements for Templates and CSElements."

## Shopping Carts and Commerce Context

CS-Direct Advantage delivers a toolset of XML and JSP tags that you can use to create a shopping cart for an online business. When a visitor starts a session on your online site, CS-Direct Advantage creates a commerce context for that session.

The CS-Direct Advantage shopping cart functions within that commerce context. You use this tag toolset to identify the contents in a cart, to calculate the price of the items in a cart, to track buyer parameters, and so on.

## GE Lighting Sample Site

CS-Direct Advantage provides a fully functional sample site named GE Lighting. The site is used in this guide as the source of examples that illustrate the flex asset model, CS-Direct Advantage functionality, and coding practices. You can examine the examples both in this book and online in the context of the actual site.

The GE Lighting sample site is an online catalog that sells lighting products. It provides two sample flex families (the product family and the content family), elements, `SiteCatalog` entries, a workflow process, and so on. For information about the GE Lighting sample site, see the following chapters:

- Chapter 11, "Data Design: The Asset Models"
- Chapter 16, "Designing Flex Asset Types"

## Custom CS-Direct Advantage XML and JSP tags

CS-Direct Advantage provides several new tag families (with both XML and JSP versions) you can use to identify, extract, and display flex assets as well as to implement an online business.

- For information about coding pages that display flex assets, see Chapter 27, "Coding Elements for Templates and CSElements."
- For information about all the custom CS-Direct Advantage tags, see the *Content Server Tag Reference*.

# Content Server Clients (Interface Options)

It also supports clients (optional interfaces) such as MS Word and other applications that offer functionality similar to Windows Explorer and Windows Desktop.

- **InSite interface** allows business users who do not ordinarily work in Content Server's interfaces but who occasionally need to approve or make changes to content to do so directly on a rendered page. The InSite interface provides access to Content Server's most commonly used content management functions, and is displayed alongside the rendered page.

  The InSite interface is automatically installed with CS-Direct. Is is also used by CS-Direct Advantage. To enable the InSite editing and page layout features, you must code your rendering templates to activate these InSite features for the appropriate asset types.

  For information about coding templates that invoke the InSite editing feature, see Chapter 35, "Coding for the InSite Editor." For information about coding templates that invoke the InSite page layout feature, see Chapter 24, "Creating Templates to Support Graphical Page Design."
- **FCKEditor** is a popular third-party WYSIWYG HTML editor from Frederico Caldeira Knabben. Developers can use FCKEditor to create basic assets whose text-entry fields use FCKEditor as the input mechanism for the field. Developers can also

create attribute editors for flex attributes that use FCKEditor as the input medium. For instructions on how to configure FCKEditor, see "Configuring FCKEditor" on page 374.

- **Clarkii Online Image Editor (Clarkii OIE)** is a popular third-party image editor from InDis Baltic. Developers can enable Clarkii OIE to allow users to edit images directly in Content Server's interfaces, eliminating the need for an external image editor. For information on how to configure Clarkii OIE on your system, see "Configuring the Clarkii Online Image Editor" on page 392.
- **Content Server Desktop (CS-Desktop)** enables content providers to use Microsoft Word, instead of the Content Server interface, to create and edit their assets. Content providers can install CS-Desktop on their personal computers. CS-Desktop then installs a Content Server toolbar into the Word client, thus providing CS-Direct functionality in the Word client.

  CS-Desktop is supported by CS-Direct. It is used by CS-Direct and CS-Direct Advantage, and must be manually installed.
- **Content Server DocLink (CS-DocLink)** provides a drag-and-drop interface for uploading and downloading documents, graphics, or other files that are managed as flex assets by Content Server. CS-DocLink presents the hierarchical data structure of the flex parents and flex assets in the Content Server database as folders and files in the Windows Explorer application.

  CS-DocLink is supported by CS-Direct, but used only by CS-Direct Advantage. It must be manually installed.
- **eWebEditPro**, a third-party HTML editor from Ektron, Inc.

For more information about configuring CS-Desktop and CS-DocLink for specific users, see the *Content Server Administrator's Guide.*

# Engage

Engage is built on top of CS-Direct Advantage and works only within the flex asset data model. This Content Server product enables your marketing team to divide your visitors into segments and then target those segments with personalized promotional, marketing, or informational messages.

Engage adds personalization and merchandising features to the Content Server product family and extends the XML and JSP tags available for programming your online site. It enables you to design online sites that gather information about your site visitors and customers, evaluate that information, and then use that information to personalize the product placements and promotional offerings that are displayed for each visitor.

Engage introduces the following concepts and features:

- Visitor data and segments
- Recommendations
- Promotions
- Persistent, linked visitor sessions
- Custom Engage XML and JSP tags

## Visitor Data and Segments

Visitor data defines the kinds of information that you want to gather about your visitors. There are three kinds of visitor data assets:

- **Visitor attributes** hold types of information that specify one characteristic only. For example, there might be attributes named “years of experience,” “job description,” or “number of children.”
- **History attributes** and **history types**. These assets create a group of related information types that you can treat as a single item. For example, an item called “purchases” could be made up of the attributes “SKU,” “itemname,” “quantity,” and “price.” The item named “purchases” is called a **history type** and the individual attributes that comprise it (price, quantity, etc.) are called **history attributes** in Engage.

**Segments** are assets that categorize groups of visitors based on the visitor data that you are gathering about them. Marketers use the visitor data assets to create segments that define groups of visitors with one or more characteristic in common: geographic location, gender, job description, item purchased, are examples.

You can define segments that are extremely broad (all first-time visitors, for example) or very focused (all first-time visitors who own RVs and live in Alaska).

- For information about creating visitor data assets, see Chapter 39, “Creating Visitor Data Assets.”
- For information about creating segment assets, see the *Content Server Advanced Interface User’s Guide*.

## Recommendations

**Recommendations** are assets that determine which flex assets (products, for example) should be featured or "recommended" on a site page. Recommendation assets are rules that are based on the segments the visitors qualify for, and, in some cases, relationships among the flex assets.

One type of recommendation, the "Dynamic List Recommendation," requires you to code a CSElement asset which returns the content that you wish to display.

Recommendations have templates. A recommendation returns a list of flex assets to its template when the template is rendered on a site page. The items in a list of recommended flex assets are rated according to their importance to the current visitor based on the segments that the visitor belongs to.

For information about creating recommendations and coding Dynamic List elements, see Chapter 40, "Recommendation Assets."

## Promotions

**Promotions** are assets that define an offer of value (a discount) based on the flex assets that the visitor is buying and the segments that the visitor qualifies for. This value can be offered in several ways:

- A discount off the purchase price of the promoted flex assets
- A discount off the entire value of the shopping cart
- A discount off shipping charges
- A combination discount: a shipping discount with a price or cart discount

All promotions can have a duration (the time period during which they are in effect). For information about creating promotions, see the *Content Server Advanced Interface User's Guide*.

## Persistent, Linked Visitor Sessions

A Content Server session ends when the visitor closes his or her browser. How, then, do you link the data that you gathered about a visitor to that same visitor when he or she returns to your online site? With a toolset of XML and JSP tags that Engage provides, called the "visitor data manager."

The visitor data manager object methods create a visitor context that enables you to link visitor sessions. You do this with persistent cookies and aliases implemented in a specific way. For information about linking visitor sessions, Chapter 41, "Coding Engage Pages."

## Custom Engage XML and JSP tags

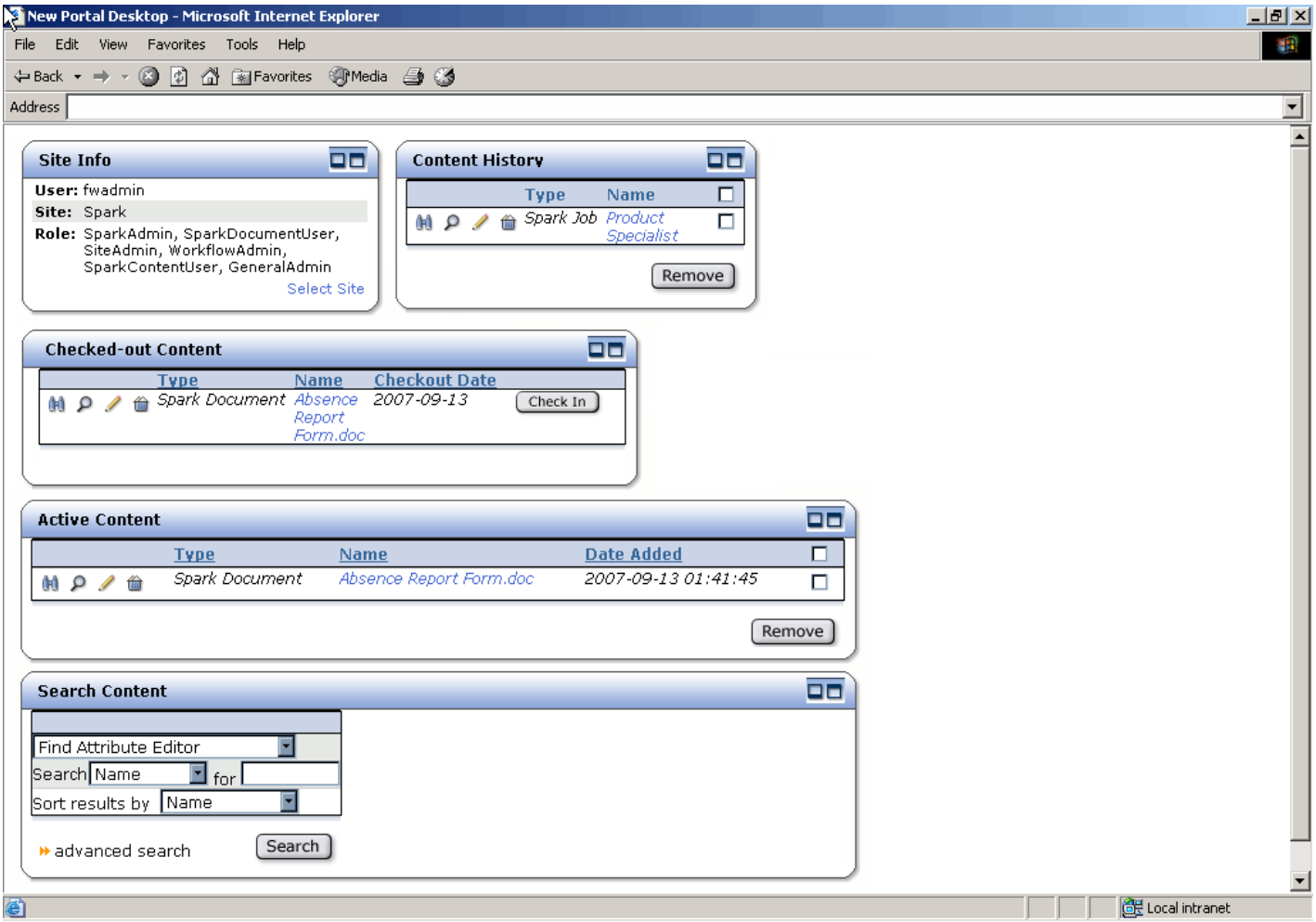
In addition to the tag family that implements the linking of visitor sessions, Engage implements several more custom tag families (both XML and JSP versions.) You use these tags to code pages that collect visitor data and to code the templates for your recommendations.

- For information about coding pages that collect and use visitor data and coding templates for recommendations, see Chapter 41, "Coding Engage Pages."
- For information about all of the Engage custom XML and JSP tags, see the *Content Server Tag Reference*.

# Content Server Portal Interface

The Content Server Portal interface enables content management operations in a portal environment.

During the installation of the Content Server product, installation engineers have the option to install the Portal interface, which displays the content providers' most common tasks and objects within portlets in a workspace area, as shown below:




The portlets, just like windows, can be minimized, maximized, and moved. Such arrangement provides for a more user-friendly experience and makes the Portal interface suitable for the less-experienced content providers.

Note that if the Portal interface is to be used, it must be installed at the same time that the Content Server product is installed.

Chapter 2

# Overview of Sites

In the Content Server environment, a “content management site” is an object that you use as an aid in the Content Server interface for designing the online site and for managing access to assets. The Burlington Financial sample site is a content management site. So is the GE Lighting sample site.

You first create content management sites on the development system. When the sites are tested and approved for use, you must then duplicate the sites (with exactly the same names) on the management and delivery systems.

This chapter contains the following sections:


- Content Management Sites
- Online Sites
- Developers and the Content Management Site
- Sites and the Site Plan
- Sites and the Database

# Content Management Sites

A **content management (CM) site** is an object that you use as a design aid or organizational construct for the online site that you are delivering from your Content Server delivery system. A content management site represents your online site.

**Note**

In this guide, *content management site* is also called "CM site," or simply "site."

When you log in to Content Server running any of the Content Server modules and products, you are logging in to a content management site. If you have access to more than one site, the first decision that you make after logging in is which site to work on. From that point on, all of the tasks that you complete are completed in the context of that site (until you switch sites).

Content management sites are used by three types of users in different ways:

- Developers use content management sites to design the online site:
  - Create the data model (the source of content-entry and editorial forms that business users will use in order to provide content for the online site)
  - Enable the data model for selected content management sites
  - Code templates that extract content from the database, format the content, and deliver the content to the online site. You use the **Site Plan** tab to create a design framework for the online site. Each content management site has a separate site plan, which is stored in the `SitePlanTree` table.
  - Enable the templates for the content management sites
  - Implement page caching
  - Make use of session data
  - Write code that gathers information about site visitors
  - Establish security

  **Note**

  When you create data (for example, asset types and templates), the data is entered into a system-wide pool, regardless of the content management site that you have chosen to log in to.

  Enabling the data for the site that you are logged in to (and for any other site) links the data to that site and makes it accessible to users of that site.

- CS administrators use content management sites to control users' access to content:
  - CS administrators can restrict users from accessing certain assets and asset types on the Content Server system.

    Asset types must be enabled for a content management site. Therefore, if an administrator decides not to enable an asset type for a content management site, then users who log in to that content management site do not have access to assets

of that type. CS administrators can also restrict users from accessing a content management site.

- CS administrators can share individual assets among content management sites (as long as the sharing sites have the asset type enabled and have the same users in common).

  CS administrators can also restrict access to specific assets by not sharing them. Even if the asset type is enabled across sites, an asset created in one site is not available in another unless it has been shared to the other site.

- Once content management sites are developed, authorized content providers use the sites to create electronic assets, manage the assets, and deploy the assets to their audiences. The content providers are linked to content-entry forms as well as other authoring tools; they are also given certain editorial permissions; and they are given access to publishing and delivery systems for serving their content as part of the online site to browsers.

Content management sites represent real, online sites. However, they can represent those online sites in any number of ways, depending on what makes sense for your situation. For example:

- One content management site can represent one complete online (public) site.
- Several content management sites can represent separate sections of one large online site. For example, with a catalog, perhaps people who do the data entry for household goods never do data entry for yard goods so there are separate sites that represent those areas. And, in a publication example, perhaps sports writers have a separate site that represents the sports news section and the financial writers have a separate site that represents the financial news section.
- Several content management sites can represent the same online site but exist to restrict users' access to asset types, by role. For example, "site one" has the article and image asset types enabled and only content providers have access to this site; "site two" has all asset types including templates enabled and only a small group of developers have access to the site.

# Online Sites

An online site is the set of pages that an organization displays to its target audience of customers, clients, and casual visitors. The online site can be an website or a portal. It can be accessible to the general public or it can be a password-protected site. It can also be a completely exclusive site, such as a corporate intranet or departmental network, operating strictly within the private domain.

Regardless of its nature, an online site originates from either a single CM site, or many CM sites, depending on which model you choose. Throughout our product guides, we use the term "online site" generically to refer to websites and portals, both of which are supported in this release.

**Note**

In this guide, *online site* is always called "online site."

# Developers and the Content Management Site

Because you must log in to a content management site when you use your Content Server modules and products, all asset development is done in the context of a content management site. As you develop asset types, design your online site pages, and code Template assets, consider the following:

- When you create a Template asset, CS-Direct creates entries for it in both the `SiteCatalog` table and the `ElementCatalog` table. The name that it assigns to the page entry in the `SiteCatalog` table includes the name of the site that you were logged in to when you created the Template asset.
- If you share a Template asset with more than one content management site, CS-Direct creates a page entry in the `SiteCatalog` for each site that it is shared with. The names of the additional page entries for a shared template include the name of the site that the template was shared with.

Therefore, you must use the same content management site names on your development system that you will use on the management and delivery systems in order for your online site to function properly.

Because content management sites cover both design issues and access issues, you must work with your system administrators when determining how to use content management sites and how many sites you need for your system.

After you determine how many content management sites you need for both design and access control reasons on your management system, you or your system administrators can create the appropriate content management sites and enable the appropriate asset types for those sites on all of your systems. Then, on your development and management systems, you or your system administrators configure which content providers and other users (such as you) have access to which sites.

To configure content management sites, you use the **Site** option on the **Admin** tab.

# Sites and the Site Plan

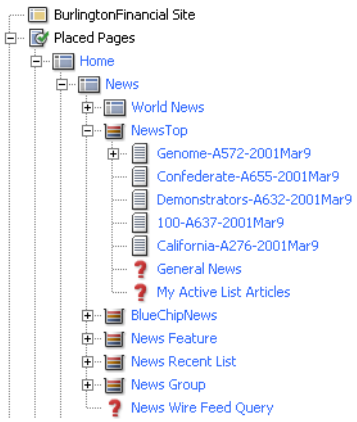
Page assets are site design assets that store references to other assets, organizing your assets according to the design that you and other developers are implementing. During the design phase of your online site, you create page assets, associate other assets with them, and then position the page assets in the tree on the **Site Plan** tab, located in the tree on the left side of the CS-Direct window.

When page assets are positioned in the tree on the **Site Plan** tab, information about each page asset's position in that tree is written to the `SitePlanTree` table. If the page assets that are positioned on this tab represent the same hierarchy that your templates and elements are coded to create on your published pages, you can use the CS-Direct `SITEPLAN` tag family to build navigational features. (See "Example 6: Displaying Site Plan Information" on page 636 for more information.)

The **Site Plan** tab displays a graphical representation of the layout of your online site, the content management site that you are currently logged in to, as a tree. It starts by querying the `SitePlanTree` table to determine which page assets have been placed. It then displays the page assets at the appropriate level on the tree, with the assets that have been associated with those page assets at subsequent hierarchical levels.

## Example: the Burlington Financial Sample Site

The tree on the **Site Plan** tab shows assets, not rendered site pages. In other words, it does not represent all the possible online pages that could be delivered by the actual online site. For example, this is the section of the **Site Plan** tree that shows the Home page asset of the Burlington Financial sample site:




To better understand the connection between your online site and the **Site Plan** tab, display the rendered Burlington Financial News page asset in your browser:

1. Log in to the Burlington Financial sample site.
2. Select the **Site Plan** tab.
3. Select the **News** page asset from the tree, click the right mouse button, and select **Preview**.

Compare the News web page that is rendered in your browser to this section of the **Site Plan** tab and note the following:

- The News page asset represents an actual page that would be rendered if a visitor selected the News link from the online Burlington Financial home page. This is because the template assigned to the News page is coded to display the page asset as a web page.
- The collection assets displayed in the tree under the News page asset do not represent actual rendered pages. The template for the News page is coded to display the headlines of the articles contained in the collections that are associated with the News page as links in the online News page.
- The article assets contained in the NewsTop collection represent actual online pages. This is because the template that displays the article when you click the link to them is coded to display the article in a separate web page.

Because it is the code in the template that determines how an asset is displayed in your online site, there can be many online pages that are not represented as page assets in the **Site Plan** tree.

In order to select the correct assets for your page assets, you must know what the template elements for your assets are coded to do. This is why creating and placing page assets is your responsibility and it is a task that you complete as you code your template elements.

Page assets serve as “gateway” or “index” pages that offer access to other assets that represent content in addition to representing actual online pages.

# Sites and the Database

When you create a site, CS-Direct writes information about it to the following database tables:

- The `Publication` table, which holds the names, descriptions, and pubids (IDs) of all the sites (publications) created for your system.
- The `PublicationTree` table, which stores information about which asset types have been enabled for which sites.
- The `SitePlanTree` table, which stores information about the hierarchical structure of a site and its page assets. There is a top-level node for each site created for your system. This table lists sites and page assets.

  As a developer, you can code your elements to extract and display information from the `SitePlanTree` table (for example, to create links to the major sections of your online site).

**Note**

Early versions of CS-Direct used the term “publication” rather than the term “site” and several of the database tables in the Content Server database still refer to sites as publications.

Chapter 3

# Content Server Development Process

When you are developing an online site that is to be delivered from a Content Server content management system, you are actually designing two sites:

- The online site that is delivered from your delivery system to visitors' browsers
- The content management site(s) that your content providers will use to input the data that they will publish to the delivery system.

In other words, you are responsible for the user experience of two sets of end users:

- The site visitors who use your delivery system
- The content providers who use the management system

When creating these two closely connected yet separate sites, the development team performs a series of planning, development, and testing steps. This chapter describes the development process in one possible sequence of events and in very general terms. Your own work flow will vary based on your work environment and business needs.

This chapter contains the following sections:


- Step 1: Set Up the Team
- Step 2: Create Functional and Design Specifications
- Step 3: Set Management System Requirements
- Step 4: Implement the Data Design
- Step 5: Build the Online Site
- Step 6: Set Up the Management System
- Step 7: Set Up the Delivery System
- Step 8: Publish to the Delivery System

# Step 1: Set Up the Team

The first step is to assemble the development team, which include the following kinds of people:

- Site designers
- XML and JSP developers
- Java application developers
- Database administrators
- System network administrators
- Marketers and advertising staff
- Product managers (if you are developing a commerce site)
- Content providers

You need people such as DBAs, system administrators, and content providers on your development team in addition to the people (like you) who do the actual coding for several reasons:

- Using a Content Server system requires you to design a data model in addition to creating a page design, which means that you need early input from the DBAs who will be supporting the databases on each system.
- Using a Content Server system means moving code and data around on multiple separate systems, several of which are probably clustered, which means you need early input from system and network administrators.
- Implementing a Content Server system will not be optimum unless the work habits of your content providers are accurately reflected in the design of the management system, which means you need early input from those who will use the management system.

# Step 2: Create Functional and Design Specifications

An online site delivered from a Content Server content management system is a holistic construct in which everything interacts, intersects, and works with everything else. Therefore, the second step is to create a functional specification and a design specification — to design your online site on paper.

You should complete some version of this step before you begin coding anything (although you might do some proof-of-concept coding while working on the design specification).

## Functional Requirements

Before you can begin a design specification, product management and marketing must provide the functional requirements for the online site.

## Page Design

After you obtain the functional requirements from your marketing folks, a good place to start is to map out all the types of pages that you want to present on the online site. For example: home page, section page, columnist page, search page, article page, and so on. If you are designing a commerce site, you need other kinds of pages: registration page, product category pages, product description page, article page, FAQ page, invoice page, and so on.

Determine the graphical, navigational, and functional features for each page and the site overall: navigation bars, buy buttons and shopping carts, “tell me more” buttons, search functions, logo placement, animated graphics, and so on.

If you are using Engage, decide where the merchandising messages (recommendations) are to be placed on the pages and on which pages they’ll be placed. For example, perhaps each product category page has a “New Products” section in the upper-right corner of the page.

Map out the entire structure of the site and create mock-ups of it.

## Caching Strategy

One of the major elements in your design is caching: page caching and resultset caching. No online site can reach performance goals without your planning, testing, and implementing a caching strategy.

While designing the pages that you want to present on your online site, you must consider how and when page caching can and should be implemented for each piece on each page.

While designing your queries, you must map out all the tables in the database and determine how the resultset caching settings should be set for each table.

## Security Strategy (Access Control)

Will you require your visitors to identify themselves before they are allowed to access any part of your online site? You must determine what kinds of access control you want to enforce early in the design process so that you design your pages correctly.

For example, if you plan to check your visitors' identities before allowing them access to a page, this affects how you would cache the components of that page. For example, you could design a container page, which is never cached, that verifies the identity of the visitor and then assembles the page from cached pagelets only if the verification is successful.

## Separate Format from Content (Elements from Assets)

Following the basic proposition of separating content from format, take a look at each piece of each proposed page in your site and determine whether that piece should be represented as data or as logic.

A good design is one in which data is designed to be represented as an asset and is not embedded into element code. Examine every component of design or content, and then determine what your assets are. You make that determination by deciding which category a component belongs to: data or logic/code.

Simply speaking, do not code something into an element (embed it in logic) if it is really data. If it is data, is should be in a separate asset.

Here's another way to look at it:

- Assets that represent content are the responsibility of content providers.
- Logic—anything coded into any element—is the responsibility of the developers.

### Determine the Asset Types (Content)

Documents, articles, products, and images are easily identified as assets. However, design components such as headers and footers could also be assets:

- When the content in a header or footer is embedded in the code of an element, you or another developer has to change the text in it when anything in it changes (a phone number, a logo, and so on).
- When the content in a header or footer is in an asset, the code in your elements must be able to obtain the identity of the asset; its content becomes the responsibility of a content provider.

Other page components that can be assets include the following kinds of things:

- Online polls
- Animation and other media
- Quote of the day
- Company or stock profiles
- Knowledgebase questions and answers

From your point of view, if the content for a component is represented in an asset, someone else is responsible for that content. You are only responsible for when and where it appears on your online sites and what it looks like when it appears there.

### Decide How to Handle Images and Other Blobs

You have two general options when deciding how to manage the images and other blobs that you want to use in your online site:

- Treat them as assets—store them in the Content Server database and have the BlobServer servlet serve them.

- Treat them as static files—put them in a file structure on your web server and let the web server serve them.

Either method is a valid option. If you keep your image files on the web server, you can create links to them with the Content Server tags, and there may be performance benefits when you allow your web server to deliver your images. However, if you keep your images and blobs separate from the Content Server database:

- You must implement a separate file management process. The publishing methods that move image assets from your management system to your delivery system cannot move content that is not in the Content Server database. You must manage this process on your own.
- None of the native Content Server security mechanisms will apply. That is, you cannot use ACLs to limit access to blobs that are not managed by Content Server.

### Map Out the Functional Design and Format (Elements)

You also need to analyze all of the functionality that you plan to incorporate into your online site. If you are designing a commerce site, parts of it will no doubt behave more like an application.

Outline what code or logic is required for your visitor registration pages, visitor data collection pages, shopping carts, personalization, and so on.

Remember that your Content Server system provides you with coding options: Java, XML, and JSP. As you look at each of the functions you want to provide, determine which is the best coding solution for that function.

## Data Design

Once you know which pieces of your site should be represented as assets, you can map out what your asset types should be. Each new asset type will use one or more database tables (depending on whether it is a basic or flex asset type).

### Asset Types

No matter which asset model you are using, basic or flex, consider the following when you design your asset types:

- Asset type design affects both of the user groups that you are designing for (visitors to the online site and the content providers who must enter the data).
- Which types of assets need to be linked or related to other assets of other types in order to successfully implement your page design? Be sure to implement these relationships in the asset type.
- Content providers appreciate efficiency. Be sure that your asset types store only the data that you really plan to use so that content providers do not waste time maintaining data that no one uses.

### Auxiliary Tables That Support Your Asset Types

The data design that you want to implement for your system extends beyond the database tables that hold your assets. Depending on the kinds of information that you want to provide, you might need to create auxiliary tables that support your asset types.

For example, the Burlington Financial sample site has asset types with a **Mimetype** field. The **Mimetype** field is a drop-down field and a user must select a value from the drop-

down list. These values are pulled from a lookup table named `MimeType`. Depending on your needs, you might need to create similar tables for your system.

Your DBAs should be involved in your discussions about the asset types and auxiliary tables that you plan to create so they can understand from the start the kind of database tuning issues that might arise on the management and delivery systems.

### Visitor Data

If you are using Engage, you also need to determine what kinds of visitor data you plan to gather. These data types are represented by the Engage visitor data assets that you use to create segments for personalizing your site based on the identity of the visitor. (For example, demographics, purchase history, or clickstream information.)

After your Content Server system goes live and you start collecting visitor data, the tables that store that data grow very quickly. This is another area that you need to consult your DBAs about.

## Step 3: Set Management System Requirements

Before you can begin coding, you must know how the management system will be organized. These decisions affect your design because your design depends on the content management site.

A content management site is an object that you use as an organizational construct for an actual online site and as an access control tool. When you create Template assets, CS-Direct creates an entry in the `SiteCatalog` table for it. The naming convention that CS-Direct uses for the page entries for templates includes the name of the content management site that you are creating the template for. This means that you must be consistent with site names throughout your entire content management system (development system, management system, and delivery system) and you must know the names of the sites that you are using before you begin coding.

Although your primary concern is the name of each site, the system administrators and business managers must also determine the following:

- How many users and ACLs (access control lists) do you need? (Remember that you may need to create ACLs to assign to the visitors of the online site, as well.)
- How many site roles you do you need?
- Which asset types need a workflow process?
- Which asset types should use revision tracking?
- Who should have access to which asset types on which sites?

Use both this book and the *Content Server Administrator's Guide* to help you make these decisions.

# Step 4: Implement the Data Design

After you have created your design specification and you understand the organization of the management system, you can implement the data design.

On the development system, you complete the following kinds of tasks:

- Create content management sites with the same names as those that will be used on the management system.
- Design and create your asset types.
- Add any lookup tables or other auxiliary tables for the asset types.
- Create sample assets of each type.

This step (step 4) and the next step (Build the Online Site) are iterative and will most likely overlap a great deal. While you need to create asset types so that you can create assets before you create templates for them, it is likely that you will uncover areas that need refinement in your data design only after you have coded a template and tested the code.

Refer to Section 3, “Data Design,” when you implement the data design of your online site.

# Step 5: Build the Online Site

After you have sample assets of even one type created on the development system, you can begin coding templates and building the online site. (Actually, you can begin coding elements that do not display assets any time after you have created your design specification.)

In this step, you complete the following kinds of tasks:

- Create the page, query, and collection assets that implement the functionality of your online site.
- If you are using Engage, create the visitor data assets, sample segments, recommendations, and sample promotions.
- Create Template assets (and code template elements) for all of your asset types.
- If you are using the InSite Editor feature, add code to the templates that invokes it.
- Code the CSElements that implement underlying functionality (that do not display assets).
- If you are developing a commerce site, code pages that implement the CS-Direct Advantage shopping cart.
- If you are using Engage, code pages that collect visitor data.
- Test everything—most likely you will perform both usability and market testing for your online site.

Refer to Section 4, “Site Development,” and Chapter 4, “Programming with Content Server” as you build your online site.

# Step 6: Set Up the Management System

After you have the online site working on your development system, you move it to the management system.

The developers complete the following kinds of tasks:

- Create the sites.
- Re-create the asset types.
- Mirror the asset type tables and auxiliary table from the development system to the management system.
- Mirror publish the site design assets and the data structure assets created on the development system to the management system.

The system administrators then complete the following kinds of tasks:

- Create users, ACLs, and roles. Assign users their roles for each content management site.
- Configure CS-Desktop users, if you are using that feature.
- Create workflow processes.
- Create StartMenu shortcuts.
- Enable revision tracking.

Refer to the *Content Server Administrator's Guide* for information about setting up the management system.

## Import Content as Assets

It is likely that you already have content in some non-asset format that you want to use. To import this content into the Content Server database as assets, use the XMLPost utility.

## Import Catalog Data and Flex Asset Data

If you are using the flex asset model and you have a large amount of pre-existing data that you want to use, you can import it with the BulkLoader utility. For systematic updates, however, you use the XMLPost utility.

## Instruct the Editorial Team About Site Design

Before the editorial team can successfully maintain the online site, they need to understand your design. For example: how frequently are collections supposed to be rebuilt?

If you are using the basic asset model, content providers need to know the following:

- Which categories and sources they should assign to their assets in order for their assets to be located by the appropriate queries and collections.
- Which templates they should assign to which assets.
- Which association fields must be filled out in order for the links on the site pages to function correctly.

It is a good idea to program as much of this information as possible into the Start Menu shortcuts that you and the system administrators create for each asset type.

If you are using the flex asset model, content providers need to know the following:

- The general hierarchy or taxonomy in place for the flex assets.
- Some information about what information a flex asset inherits.
- Which templates they should assign to which assets.

# Step 7: Set Up the Delivery System

When you set up the delivery system, you complete several of the same steps that you complete for the management system. For example:

- Re-create the sites.
- Re-create the asset types (but without their Start Menu shortcuts).
- Mirror the asset type tables and auxiliary table from the development system to the management system.

And then you publish all of the assets on the management system to the delivery system.

Also, because this system is not a management system, you complete the following steps as well:

- Implement your security strategy.
- On the web server, map the URL of your site (`www.`*`yourcompany`*`.com`) to the Content Server URL of your home page.

For information about setting up the delivery system, see the section on publishing in the *Content Server Administrator's Guide*.

# Step 8: Publish to the Delivery System

When the content on the management system is ready, you publish it to the delivery system. After intensive testing—both performance and load—you open your site to the public.

Part 2

# Programming Basics

This part provides basic programming information for coding online sites using the toolsets delivered with the Content Server products.

It contains the following chapters:


- Chapter 4, “Programming with Content Server”
- Chapter 5, “Page Design and Caching”
- Chapter 6, “Intelligent Cache Management with Content Server”
- Chapter 7, “Advanced Page Caching Techniques”
- Chapter 8, “Content Server Tools and Utilities”
- Chapter 9, “Sessions and Cookies”
- Chapter 10, “Error Logging and Debugging”

Chapter 4

# Programming with Content Server

In addition to managing your content, Content Server handles many useful tasks for you, including storing web pages and pieces of web pages—called **pagelets**— in Content Server caches, and maintaining those caches so that visitors to your website never see an outdated page. In order for Content Server to do this, you must code with Content Server tags and Java methods.

A Content Server page is composed of various **element assets**—blocks of code that can retrieve the content of your pages from the Content Server database, or that perform other tasks, such as deleting outdated items from the database—and **Template assets**, which are generally used to format the content of your web pages. Elements and templates can be written in a number of scripting and markup languages, including HTML, XML, JSP, CSS, and JavaScript. Note, however, that Content Server only evaluates XML and JSP.

This chapter gives you a brief overview of programming with Content Server. It contains the following sections:


- Choosing a Coding Language
- The Content Server Context
- Content Server JSP
- Content Server XML
- Content Server Tags
- Variables
- Other Content Server Storage Constructs
- Values for Special Characters

# Choosing a Coding Language

Choose your coding or markup language based on what the element or template that you are creating does. For example, you typically use HTML and XML for page layout and JSP and Java for logic. Elements that display content that may change, such as a newspaper article, should usually be written in XML or JSP. This is because such elements use logic to retrieve their content from the Content Server database, and thus are managed using Content Server XML or JSP tags.

Content Server also has a Java API, which you will use in conjunction with Content Server JSP tags if you choose JSP as your coding language.

The following table lists the situations to which each language is best suited:

| Code | When to Use |
|---|---|
| XML | • The element contains mostly text, with few loops and conditionals. |
| JSP | • The element requires conditional operators, or relational operators other than `=` or `!=`.<br>• The element uses many loops. Loops perform better in JSP than in XML.<br>• The element contains calls to Java code. |

Note that elements written in XML or JSP can call any type of element, but you cannot mix XML and JSP in the same element. For example, an element written in either XML or JSP can call another element written in HTML, XML, or JSP. However, an element written in HTML cannot call an element written in XML or JSP.

# The Content Server Context

When you code for a Content Server project, you code within the Content Server context. The Content Server context provides access to the Java servlets that compose Content Server, and to the Content Server Java objects whose methods and tags allow you access to Content Server functionality.

You code in the Content Server context no matter what language you code your project in; Content Server XML and JSP tags provide an easy-to use interface to Content Server's Java objects, so that even web designers with little or no Java experience can create Content Server web pages.

## The ICS Object

When you are coding for Content Server, you often access the methods and tags of the Interface to Content Server (ICS) object. The ICS object encapsulates some of Content Server's core functionality, allowing you to access servlets that control the Content Server tree (the TreeManager servlet) and the input of data into the database (the CatalogManager servlet).

You also use ICS methods and tags to perform tasks such as creating and displaying variables and using `if/then` statements to perform tasks based on specified conditions. For a complete list of the ICS object's methods and tags, see the *Content Server Tag Reference.*

## The FTCS tag

Each Content Server element or template begins and ends with the `ftcs` tag. This tag creates the Content Server context, alerting Content Server that code contained within the opening and closing `ftcs` tags will contain Content Server tags and access ICS methods.

If you use the Content Server user interface or the Content Server Explorer tool to create elements and templates, the opening and closing `ftcs` tags are automatically added after the standard directives. You must code within the opening and closing `ftcs` tags; Content Server is unaware of any code which falls outside of these tags.

If you create element and template code using some other method, you must add the opening `ftcs` tag after your directives, and use the closing `ftcs` tag as the last line of your code.

# Content Server JSP

JSP programmers have a set of standard tools at their disposal, including directives, actions, and JSP objects. If you are programming in JSP within Content Server, you have access to many of these features. Sometimes, however, you must substitute a Content Server tag for a JSP directive or action, or access a Content Server object rather than one of JSP's implicit objects.

The following sections detail the differences between standard JSP and Content Server JSP, and how standard JSP functionality maps to Content Server tags and methods:

- Content Server Standard Beginning
- JSP Implicit Objects
- Syntax
- Actions
- Declarations
- Scriptlets and Expressions
- JSP Directives
- Content Server Tag Libraries

## Content Server Standard Beginning

If you use either the Content Server user interface or Content Server Explorer to create your Template assets, CSElement assets, and non-asset elements, Content Server automatically seeds the element or template with a standard beginning.

The standard beginning for a JSP element in Content Server Explorer follows:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%@ taglib prefix="ics" uri="futuretense_cs/ics.tld" %>
<%@ taglib prefix="satellite" uri="futuretense_cs/satellite.tld"
%>
<%//
// elementName
//
// INPUT
//
```

```
// OUTPUT
//%>
<%@ page import="COM.FutureTense.Interfaces.FTValList" %>
<%@ page import="COM.FutureTense.Interfaces.ICS" %>
<%@ page import="COM.FutureTense.Interfaces.IList" %>
<%@ page import="COM.FutureTense.Interfaces.Utilities" %>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<cs:ftcs>

<!-- user code here -->

</cs:ftcs>
```

If you use the Content Server user interface to create Template and CSElement assets, you will also see a standard beginning similar to the preceding code sample. The standard beginning for these assets imports additional tag libraries for use with basic assets and includes tags that log dependencies between the Template and CSElement assets and the content that they render.

If you use a tool other than Content Server Explorer or the Content Server user interface to create your elements and templates, you must copy the standard beginning into your code verbatim.

The following sections explain the standard beginning for Content Server Explorer.

## Taglib Directives

The following taglib directives import the base tag libraries that you will use with Content Server. If you use the Content Server user interface to create template and CSElement assets, you will see additional taglib directives in your seed code.

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%@ taglib prefix="ics" uri="futuretense_cs/ics.tld" %>
<%@ taglib prefix="satellite" uri="futuretense_cs/satellite.tld"
%>
```

The first directive imports the `ftcs1_0` tags, which create the FTCS context. These tags are used in each template or element that you create, and indicate that the code enclosed by them will be controlled by Content Server.

The second directive imports the `ics` tags, which provide access to Content Server's core functionality.

The third directive imports the satellite tags, which are for use with Satellite Server.

For more information about these tag libraries, see "Content Server Tag Libraries" on page 87.

For information about commonly used tags that are found in these tag libraries, see "Content Server Tags" on page 92.

To add taglib directives to these defaults, modify and save the `OpenMarket/Xcelerate/AssetType/Template/ModelJsp.xml` file.

## Page Directives

The following page directives import the base Java interfaces that you will use with Content Server:

```
<%@ page import="COM.FutureTense.Interfaces.FTValList" %>
<%@ page import="COM.FutureTense.Interfaces.ICS" %>
<%@ page import="COM.FutureTense.Interfaces.IList" %>
<%@ page import="COM.FutureTense.Interfaces.Utilities" %>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
```

The first page directive imports the FTValList interface, which creates a list of name/value pairs that you use to pass arguments to Content Server subsystems like the CatalogManager and TreeManager.

The second page directive imports the `ICS` interface, which provides access to the core Content Server functionality.

The third page directive imports the IList interface, which contains the methods to access the rows in a Content Server query or list object. It also contains the methods that a third party must implement when attempting to construct and register a list object for use within an Content Server XML page.

The fourth page directive imports the `Utilities` interface, which provides a simple interface for some common tasks such as formatting dates, reading and writing files, and sending e-mail.

The fifth page directive imports the ftErrors class, which contains error codes.

The sixth page directive imports the `ftMessage` class, which contains error messages used by Content Server.

To add page directives to the standard directives for JSP elements, modify and save the `OpenMarket/Xcelerate/AssetType/Template/ModelJsp.xml` file.

## The cs:ftcs Tag

Each Content Server JSP template or element must have the cs:`ftcs` tag as its first and last tags. This tag creates the Content Server context, alerting Content Server that code contained within the opening and closing `cs:ftcs` tags will contain Content Server tags.

You must code within the opening and closing `cs:ftcs` tags; Content Server is unaware of any code which falls outside of these tags.

## JSP Implicit Objects

JSP provides several implicit objects that are available for developers to use. In the Content Server context, however, you are often dealing with Content Server's objects, and should use Content Server JSP tags and Java methods to access these objects, instead of using JSP's implicit objects.

The following table maps JSP's implicit objects and some of their commonly used methods to the Content Server tag or method that you should use to replace them.

| Object | Method | Content Server Tag or Method |
|---|---|---|
| `request` | `getParameter` | `ics:getvar` tag |
| | `getParameterNames` | `ICS.GetVars()` method |
| | `getCookie` | `ics:getCookie` tag |
| `response` | `addCookie` | `satellite:cookie` tag |
| `session` | `getAttribute` | `ics:getssvar` tag |
| | `setAttribute` | `ics:setssvar` tag |
| `out` | `println` | `ics:getvar` tag or `render:stream` tag |

## Syntax

Content Server uses standard JSP syntax. When you are nesting tags, however—for example, using a JSP expression as the value of a JSP tag's parameter—remember to use single quotes to contain the expression, as in the following example:

```
name='<%=ics.GetVar("myVariable")%>'
```

## Actions

Standard JSP allows developers to use several different actions. The following table describes what actions should be replaced with Content Server tags and which can be used as usual:

| Action | Content Server |
|---|---|
| `<jsp:forward>` | Use the `render:satellitepage` or `render:callelement` tags instead. |
| `<jsp:getproperty>` | Use this for custom Java Beans. If you want to find the value of one of the Content Server properties, use the `<ics:getproperty>` tag. |
| `<jsp:include>` | Use the `render:satellitepage` or `render:callelement` tags instead. |
| `<jsp:setProperty>` | Use this to set properties in custom Java Beans. Use the Content Server Property Editor to set Content Server properties. |
| `<jsp:useBean>` | Use this for custom Java Beans. |

## Declarations

In standard JSP, you usually declare variables within a JSP declaration. In Content Server, you use the `ics:setvar` tag to declare variables that are available in the Content Server context.

For more information about Content Server variables, see "Variables" on page 104.

## Scriptlets and Expressions

You can use scriptlets and expressions without any variation from normal JSP usage.

When you use an expression as the value of the parameter for a Content Server JSP tag, however, be sure that you nest quotation marks correctly, as described in "Syntax" on page 86.

## JSP Directives

When you are coding JSP in a Content Server context, there are some caveats for using directives, which are outlined in the following table:

| Directive | Content Server |
|---|---|
| IncludeDirective | Use the `render:satellitepage` or `render:callelement` tags to include other files in your JSP pages. |
| Page Directive | If you use the Content Server user interface or the Content Server Explorer tool to create elements or templates, your element or template is automatically seeded with standard page directives.<br>In addition to the standard directives, you must add one other page directive to set the `contentType` for each Content Server element or template that you create.<br>Set your page's content type to text/html and the character set to UTF-8 by providing the following page directive as the first line of every Content Server JSP file:<br>`<%@ page contentType="text/html; charset=UTF-8" %>` |
| Taglib Directive | Content Server automatically seeds your templates and elements with commonly used taglib directives.<br>You can add additional Content Server taglib directives to an element or Template asset as needed; a list of the Content Server tag libraries follows this table. |

## Content Server Tag Libraries

Content Server has a series of JSP tag libraries that correspond to functions in Content Server's APIs.

The following table lists the Content Server tag libraries and describes their functions. Use this table as a reference when deciding which tag libraries to import into your JSPs.

## Tag Libraries for Both Basic and Flex Assets

| Tag Library | Description |
|---|---|
| `acl.tld` | Tags for creating and manipulating Access Control Lists. |
| `date.tld` | Tags that convert dates with year, month, day, and optional hour, minute, and am/pm fields into epoch format long integers representing milliseconds since Jan 1, 1970, 0:00 GMT. Date tags also convert long integers into dates. |
| `dir.tld` | Directory Services tags. |
| `ftcs1_0.tld` | Tags that create the FTCS context. These tags are used in each template or element that you create, and indicate that the code enclosed by them will be controlled by Content Server. |
| `ics.tld` | Tags which provide access to core Content Server functionality, including access to the CatalogManager and TreeManager commands, and basic coding constructs like if/then statements. |
| `insite.tld` | Tags for InSite Editor. |
| `localestring.tld` | Tags for localizing text strings. |
| `name.tld` | Tags that access the name of the user who is currently logged in to Content Server and manipulate usernames in directory services. |
| `object.tld` | Tags for manipulating Content Server objects. |
| `property.tld` | Tags for retrieving values from Content Server property files. |
| `render.tld` | Tags that render basic assets. |
| tags for working with satellite server | Many of these tags have RENDER equivalents (as defined in `render.tld`) that are preferred for building sites with CS-Direct. |
| `soap.tld` | Content Server SOAP tags. |
| `time.tld` | Tags that get and set the timing for determining the performance of elements. |
| `user.tld` | Tags to log users in and out of Content Server. |
| `webservices.tld` | Web services tags that allow you to consume certain types of public web sites as part of a Content Server page. |

## Tag Libraries for Basic Assets

| Tag Library | Description |
|---|---|
| `asset.tld` | Tags that retrieve and manipulate basic assets. |

| | |
|---|---|
| `siteplan.tld` | Tags that allow access to the site plan tree. You use these tags to create navigation for a site that uses basic assets. |

## Tag Libraries for Flex Assets

| Tag Library | Description |
|---|---|
| `assetset.tld` | Tags for creating assetsets with flex assets. |
| `blobservice.tld` | Tags for retrieving and manipulating blobs that are attributes of flex assets. |
| `calculator.tld` | Tags that provide basic calculator and boolean functions. |
| `cart.tld` | Tags that allow you to add, delete, and otherwise manipulate items in a shopping cart object. |
| `cartset.tld` | Tags that allow you store, retrieve, delete, and list shopping cart objects for a registered buyer. |
| `commercecontext.tld` | Tags that access the objects in the CS-Direct Advantage visitor context. |
| `currency.tld` | Tags that convert floating point values and currency strings, and perform formatting and rounding operations on currency strings. |
| `decimal.tld` | Tags that format floating point values as decimal objects in different locales. |
| `hash.tld` | Tags that allow you to cast an IList as a hash table and search it by key. |
| `listobject.tld` | Tags that construct Content Server resultset lists, which are used throughout your elements as arguments for other CS-Direct Advantage tags. |
| `locale1.tld` | Tags that generate a locale object, which is used to describe the desired locale for various other tags in the system. |
| `misc.tld` | Miscellaneous tags, including a tag that returns the names of all the columns in an input list |
| `searchstate.tld` | Tags for creating searchstates to constrain groups of flex assets (assetsets). |
| `session.tld` | A tag that flushes all stored objects for a given session. |
| `string.tld` | Tags that perform string manipulations. |
| `textformat.tld` | Tags that format text. |
| `vdm.tld` | Visitor Data Management tags, which enable you to record and retrieve information about website visitors from Content Server, or from other databases. |

For complete descriptions of the CS-Direct tags used for template development, see the *Content Server Tag Reference*.

# Content Server XML

This section explains the basics of Content Server XML. Content Server XML uses standard XML syntax and is defined by the `futuretense_cs.dtd`. As with Content Server JSP tags, Content Server XML tags provide access to Content Server servlets and objects.

The following sections describe things to be aware of when coding with Content Server XML.

## Content Server Standard Beginning

If you use the Content Server user interface or the Content Server Explorer tool to create your templates and elements, Content Server automatically seeds the element with the following standard beginning:

```
<?xml version="1.0" ?>
<!DOCTYPE ftcs SYSTEM "futuretense_cs.dtd">
<ftcs version="1.2">
</ftcs>
```

If you use some other tool to create your elements and templates, you must copy this code into them verbatim.

The following sections explain this standard beginning.

### XML Version and Encoding

The first line in any Content Server XML template or element must set the XML version, as follows:

```
<?xml version="1.0"?>
```

Note that in order for your element to run, `<?xml version="1.0"?>` must be the first line in the element, with no spaces before the text. The line must also have a hard return at the end, placing it on its own line.

If you need to set the encoding for this template or element, you can do this as follows:

```
<?xml version="1.0" encoding="utf-8"?>
```

### The .dtd File

Content Server XML is defined by the `futuretense_cs.dtd` file. You must import this file into each Content Server element or template that you code by entering the following line immediately after the XML version statement:

```
<!DOCTYPE ftcs SYSTEM "futuretense_cs.dtd">
```

### The FTCS Tag

Each Content Server XML template or element must have the `ftcs` tag as its first and last tags. This tag creates the Content Server context, alerting Content Server that code contained within the opening and closing `FTCS` tags will contain Content Server tags.

You must code within the opening and closing `ftcs` tags; Content Server is unaware of any code which falls outside of these tags.

## XML Entities and Reserved Characters

Because symbols such as < and > are reserved characters in XML, you must not place them in your content. For example, the following code confuses the XML parser because the less-than sign (<)appears inside some text:

```
<P>4 < 7</P>
```

You must use character entities in place of reserved characters. Character entities begin with `&#` and end with a semicolon. Between the `&#` and the semicolon, you specify the decimal Latin-1 (a superset of ASCII) value of the character. For example, the decimal Latin-1 value of the `<` character is 60, so the correct way to code the preceding line in XML is:

```
<P>4 &#60; 7</P>
```

See the "Values for Special Characters" section of this chapter for a list of these character entities.

## XML Parsing Errors

The XML parser that processes Content Server tags ensures that the tags are syntactically correct. This simplifies tracking down hard-to-find problems related to tagging syntax errors. A misspelled tag name is not reported as an error. This is because the XML parser doesn't require all tag names to exist in the DTD.

When a page request is made to a Content Server system and an XML syntax error is detected, the results streamed back will contain useful information to help you locate the problem. The results include a general error description, followed by the line/column location of the error. For example, the following error reports a bad parameter name:

```
Illegal attribute name NAM Illegal attribute name NAM
Location: null(6,11)
Context:
```

And the next error reports an incorrect tag nesting:

```
Close tag IF does not match start tag THEN Close tag IF does
not match start tag THEN
Location: null(13,3)
Context:
```

The XML parser also detects run-time errors. These are errors where the XML tags are syntactically correct, however, some error in the structure is detected during processing. For example, the following error reports an invalid use of `ARGUMENT`:

```
Failed to run template:c:\FutureTense\elements\dan.xml Runtime
error Argument invalid [Argument 5]
Containing tag: FTCS
```

# Content Server Tags

Content Server has an extensive set of tags in both JSP and XML that allow you to access the various functions of Content Server and its product family. You use these tags in conjunction with HTML, Java, JavaScript, and custom tags that you create, to code your web site.

This section provides and overview of the tags that you are most likely to use in your Template assets and elements. For complete information on all of the Content Server tags, see the *Content Server Tag Reference*.

The tags discussed here are arranged by usage, as follows:

- Tags That Create the Content Server Context
- Tags That Handle Variables
- Tags That Call Pages and Elements
- Tags That Create URLs
- Tags That Control Caching
- Tags That Set Cookies
- Programming Construct Tags
- Tags That Manage Compositional and Approval Dependencies
- Tags That Retrieve Information About Basic Assets
- Tags That Create Assetsets (Flex Assets)
- Tags That Create Searchstates (Flex Assets)

## Tags That Create the Content Server Context

The following table describes tags that create the Content Server context in which you code. You use these tags in every template or element that you write.

| FTCS (XML) | ftcs1_0:ftcs (JSP) |
|---|---|
| `<FTCS>`<br>`</FTCS>` | `<ftcs1_0:ftcs>`<br>`</ftcs1_0:ftcs>` |

The FTCS tag creates the Content Server context. The opening FTCS tag should be the first tag in your code, and the closing FTCS tag should be the last tag in your code. Content Server is unaware of anything that falls outside of the opening and closing FTCS tags. Consequently, content outside the tags will not be cached, and the tags will not operate correctly.

## Tags That Handle Variables

The following tags handle variables in Content Server.

| CSVAR (XML) | ics:getvar (JSP) |
|---|---|
| `<CSVAR`<br>`NAME="variableName"/>` | `<ics:getvar`<br>`name="variableName"/>` |

`CSVAR` displays the value of a variable, session variable, built-in, or counter.

| SETVAR (XML) | ics:setvar (JSP) |
|---|---|
| `<SETVAR`<br>`NAME="variableName"`<br>`VALUE="variableValue"/>` | `<ics:setvar`<br>`name="variableName"`<br>`value="variableValue"/>` |

`SETVAR` sets the value of a regular, Content Server variable.The value of the variable exists for the duration of the page evaluation unless it is explicitly deleted using `REMOVEVAR`.

| SETSSVAR (XML) | ics:setvar (JSP) |
|---|---|
| `<SETSSVAR`<br>`NAME="variableName"`<br>`VALUE="variableValue"/>` | `<ics:setssvar`<br>`name="variableName"`<br>`value="variableValue"/>` |

`SETSSVAR` sets a session variable.

| REPLACEALL (XML) | ics:resolvevariables (JSP) |
|---|---|
| `<REPLACEALL`<br>`NAME="variableName"`<br>`VALUE="variableValue"/>` | `<ics:resolvevariables`<br>`name="variableName"`<br>`[output="variable name"]`<br>`[delimited="true\|false"]/>` |

`REPLACEALL` and `ics:resolvevariables` resolve multiple Content Server variables. In other words, when you want to use Content Server variables in HTML tags, you use these tags to resolve the variables.

For more information about variables in Content Server, see "Variables" on page 104.

## Tags That Call Pages and Elements

Use the following tags to call elements or templates.

**Note**

CACHECONTROL, used below, has been deprecated.

| RENDER.SATELLITEPAGE (XML) | render:satellitepage (JSP) |
|---|---|
| `<RENDER.SATELLITEPAGE`<br>`PAGENAME="nameOfPageEntry"`<br>`[CACHECONTROL="expiration_date_and_time"]`<br>`[ARGS_var1="value1"]/>` | `<render:satellitepage`<br>`pagename="nameOfPageEntry"`<br>`[cachecontrol="expiration_date_and_time"]>`<br>`<[render:argument name="variable1" value="value1"]/>`<br>`</render:satellitepage>` |

`RENDER.SATELLITEPAGE` requests a Content Server pagelet and caches that pagelet in both Content Server and Satellite Server, if the pagelet is not already in cache. If you wish to call a page or pagelet without caching it individually, use the `RENDER.CALLELEMENT` tag. The `RENDER.SATELLITEPAGE` tag has a stacked scope, so the only variables available to the page are ones that you explicitly pass in.

| RENDER.CALLELEMENT (XML) | render:callelement (JSP) |
|---|---|
| `<RENDER.CALLELEMENT`<br>`ELEMENTNAME="nameOfElement"`<br>`[ARGS_var1="value"]/>` | `<ics:callelement`<br>`element="element name">`<br>`<ics:argument name="argument name" value="arg value"/>`<br>`</ics:callelement>` |

`RENDER.CALLELEMENT` is similar to the `RENDER.SATELLITEPAGE` tag in that both tags call other Content Server code, either in an element or in a page. However, code called by `RENDER.CALLELEMENT` does not get cached as an individual page or pagelet on Satellite Server.

Use RENDER.CALLELEMENT to process the content of an element that you wrote for the CS Content Applications and you want the scope of that element to be stacked. The element must exist in the ElementCatalog.

## Tags That Create URLs

| RENDER.GETPAGEURL (XML) | render:getpageurl (JSP) |
|---|---|
| <RENDER.GETPAGEURL | <render:getpageurl |
| OUTSTR="*myURL*" | outstr="*myURL*" |
| PAGENAME="*SiteCatalogPageEntry*" | pagename="*SiteCatalogPageEntry*" |
| cid="*IDofAsset*" | cid="*IDofAsset*" |
| [p="*IDofParentPage*"] | [p="*IDofParentPage*"] |
| [c="*AssetType*"] | [c="*AssetType*"] |
| [ADDSESSION="*true*"] | [addsession="*true*"] |
| [DYNAMIC="*true*"] | [dynamic="*true*"] |
| [PACKEDARGS="*stringFromPACKARGS tag*"] | [packedargs="*stringFromPACKARGStag*"] |
| [ARGS_*xxx*="*y*"]/> | <[render:argument name="*xxx*" value="*yyy*"]/> |
| | </render:getpageurl> |

This tag creates a URL for an asset, processing the arguments passed to it into a URL-encoded string and returning it as the variable specified by the OUTSTR parameter. If rendermode is set to export, the tag creates a file name for a static HTML file (unless you specify that you want a dynamic URL). If rendermode is set to live, the tag creates a dynamic URL.

| RENDER.SATELLITEBLOB (XML) | render:satelliteblob (JSP) |
|---|---|
| <RENDER.SATELLITEBLOB | <render:satelliteblob |
| SERVICE="*HTMLtagName*" | service="*HTMLtagName*" |
| BLOBTABLE="*blobTable*" | blobtable="*blobTable*" |
| BLOBKEY="*primaryKeyName*" | blobkey="*primaryKeyName*" |
| BLOBWHERE="*primaryKeyValue*" | blobwhere="*primaryKeyValue*" |
| BLOBCOL="*columnName*" | blobcol="*columnName*" |
| BLOBHEADERNAMEN="*headername*" | blobheadernameN="*headername*" |
| BLOBHEADERVALUEN="mimetype" | blobheadervalueN="mimetype" |
| [ARGS_format1="*5*"] | [cachecontrol="*expirationDateAndTime*"]> |
| [CACHECONTROL="*expirationDateAndTime*"]/> | <[render:argument name="*format1*" value="*5*"]/> |
| | </render:satelliteblob> |

This tag creates an HTML tag with a BlobServer URL for assets that are blobs. For example, imagefile assets from the Burlington Financial sample site are blobs stored in the Content Server database which means they must be served by the BlobServer servlet. This

tag creates an HTML tag that instructs a browser how to find and format the specified blob.

## Tags That Control Caching

The following tag allows you to control whether or not the output of the current template or element gets cached.

| ics.disablecache (XML) | ics:disablecache (JSP) |
|---|---|
| `<ics.disablecache/>` | `<ics:disablecache/>` |

Use `ics.disable` cache in conjunction with `if/then` statements that check for error conditions; if an error is present, the resulting rendered page will not be cached. For complete information and code samples for the `ics.disablecache` tag, see "Ensuring that Incorrect Pages Are Not Cached" on page 617.

## Tags That Set Cookies

The following tag sets cookies in Content Server.

| satellite.cookie (XML) | satellite:cookie (JSP) |
|---|---|
| `<satellite.cookie`<br>`name="cookie_name"`<br>`value="cookie_value"`<br>`timeout="timeout"`<br>`secure="true\|false"`<br>`url="URL"`<br>`[domain="domain"]/>` | `<satellite:cookie>`<br>`<satellite:parameter name='name'`<br>`value='cookie_name'/>`<br>`<satellite:parameter name='value'`<br>`value='cookie_value'/>`<br>`<satellite:parameter name='timeout'`<br>`value='cookie_timeout'/>`<br>`<satellite:parameter name='secure'`<br>`value='true\|false'/>`<br>`<satellite:parameter name='url'`<br>`value='url'>`<br>`</satellite:cookie>` |

`satellite.cookie` sets a cookie on the user's browser. This tag is the only way to set cookies in either XML or JSP.

## Programming Construct Tags

The following tags allow you to use basic programming constructs.

| IF/THEN/ELSE (XML) | ics:if/ics:then/ics:else (JSP) |
|---|---|
| `<IF COND="LOGICAL_EXPRESSION">`<br>`<THEN>`<br>`tags and/or text`<br>`</THEN>`<br>`<ELSE>`<br>`tags and/or text`<br>`</ELSE>`<br>`</IF>` | `<ics:if condition="logical expression">`<br>`<ics:then>`<br>`tags and/or text`<br>`</ics:then>`<br>`<ics:else>`<br>`tags and/or text`<br>`</ics:else>`<br>`</ics:if>` |

`IF`, `THEN`, `ELSE` determine conditions. You typically use these tags to determine the value of a variable.

| LOOP (XML) | ics:listloop (JSP) |
|---|---|
| `<LOOP [FROM="START"]`<br>`[COUNT="LOOP_TIMES"]`<br>`[LIST="LIST_NAME"]`<br>`[UNTIL="END"]>`<br>`...`<br>`</LOOP>` | `<ics:listloop`<br>`listname="some list"`<br>`[maxrows="number of loops"]`<br>`[startrow="start row"]`<br>`[endrow="end row"]/>` |

`LOOP` and `ics:listloop` iterate through items in a list. Remember that excess code within these tags affects the performance of the template. Whenever possible, keep statements that do not need to be repeated outside the `LOOP` tags.

## Tags That Manage Compositional and Approval Dependencies

For complete information about compositional and approval dependencies, see "About Dependencies" on page 580.

| RENDER.LOGDEP (XML) | render:logdep (JSP) |
|---|---|
| `<RENDER.LOGDEP ASSET="asset name"`<br>`CID="asset id"`<br>`C="asset type"/>` | `<render:logdep asset="asset name"`<br>`cid="asset id"`<br>`c="asset type"/>` |

Use the `RENDER.LOGDEP` tag if your template uses tags that obtain an asset's data without loading the asset, such as `ASSET.CHILDREN`.

| RENDER.UNKNOWNDEPS (XML) | render.unknowndeps (JSP) |
|---|---|
| `<RENDER.UNKNOWNDEPS/>` | `<render:unknowndeps/>` |

Use the `RENDER.UNKNOWNDEPS` tag if a page has a query or some other indeterminate connection to its dependent assets. This tag causes the page or pagelet to be regenerated at every publish because the dependencies cannot be determined. This means that you should use this tag sparingly.

| RENDER.FILTER (XML) | render:filter (JSP) |
|---|---|
| `<RENDER.FILTER LIST="list name"`<br>`LISTVARNAME="output list name"`<br>`LISTDICOL="assetID column"`<br>`[LISTTYPECOL="assettype column"]`<br>`[TYPE="asset type"]`<br>`[ID="asset id"]`<br>`[VARNAME="output variable"/>` | `<render:filter list="list name"`<br>`listvarname="output list name"`<br>`listidcol="assetID column"`<br>`[listtypecol="assettype column"]`<br>`[type="asset type"]`<br>`[id="asset id"]`<br>`[varname="output variable"/>` |

Use the `RENDER.FILTER` tag to check for unapproved assets and prevent them from being included in the exported page. This tag filters either a single asset or list of assets by comparing each asset ID against the `assetid` column in the `ApprovedAssets` database table. During export rendering, it filters what can be published based on approval status. During live rendering, `RENDER.FILTER` does nothing. Use this tag whenever you have a database query for a list of assets in your template.

## Tags That Retrieve Information About Basic Assets

| ASSET.LOAD (XML) | asset:load (JSP) |
|---|---|
| `<ASSET.LOAD`<br>`NAME="`*`assetName`*`"`<br>`TYPE="`*`assetType`*`"`<br>`OBJECTID="`*`object.id`*`"`<br>`[FIELD="`*`fieldName`*`"]`<br>`[VALUE="`*`fieldValue`*`"]`<br>`[DEPTYPE="EXACT, EXISTS, or GREATER"]/>` | `<asset:load`<br>`name="`*`assetName`*`"`<br>`type="`*`assetType`*`"`<br>`objectid="`*`object.id`*`"`<br>`[field="`*`fieldName`*`"]`<br>`[value="`*`fieldValue`*`"`<br>`[deptype="exact,exists,or greater"]/>` |

This tag queries the database for a specific asset and then loads the asset's data into memory as an object. The object is then available to your elements until either the session is flushed or the name that is assigned to the object is overwritten.

The scope of the object names that you assign to loaded assets is **global.** Be sure to use unique object names so that your elements do not overwrite objects by mistake. A convenient naming convention is to include the element name in the asset name. For an example of creating unique asset object names by using this convention, see "Example 1: Basic Modular Design" on page 620.

ASSET.LOAD automatically logs a dependency between the template or element that uses the tag and the asset data that the tag retrieves.

| ASSET.SCATTER (XML) | asset:scatter (JSP) |
|---|---|
| <ASSET.SCATTER<br>NAME="*assetName*"<br>PREFIX="*variablePrefix*"/> | <asset:scatter<br>name="*assetName*"<br>prefix="variablePrefix"/> |

This tag retrieves values from all of the fields of an asset object that has already been retrieved (loaded) with the ASSET.LOAD tag and turns those values into Content Server variables. For example, if you want to display the headline, byline, description, and so on of an article online, you can use this tag to retrieve all of those values with one call.

| ASSET.GET (XML) | asset:get (JSP) |
|---|---|
| <ASSET.GET<br>NAME="*assetName*"<br>FIELD="*fieldName*"<br>[OUTPUT="outputVariable"]/> | <asset:get<br>name="assetName"<br>field="fieldName"<br>[output="outputVariable"]/> |

This tag retrieves the value from one specified field of an asset object that has already been retrieved (loaded) with the ASSET.LOAD tag and turns that value into a Content Server variable. For example, if you need only the headline of an article to use in a link to that article, you can use this tag to retrieve that one value.

| ASSET.CHILDREN (XML) | asset:children (JSP) |
|---|---|
| <ASSET.CHILDREN<br>NAME="*assetName*"<br>LIST= "*listName*"<br>[CODE= "*NameOfAssociation*"]<br>[OBJECTTYPE= "*typeOfObject*"]<br>[OBJECTID="*objectID*"]<br>[ORDER="*nrank*"]/> | <asset:children<br>name="*assetName*"<br>list="*listName*"<br>[code="*NameOfAssociation*"]<br>[objectype="*typeOfObject*"]<br>[objectid="*objectID*"]<br>[order="*nrank*"]/> |

This tag queries the AssetRelationTree table and then builds a list of assets that are children of the asset that you specified. You use this tag to retrieve assets in a collection, to retrieve the image assets associated with article assets, and so on.

Use the RENDER.LOGDEP tag in conjunction with ASSET.CHILDREN to log a dependency between the element or template in which it appears and the content that ASSET.CHILDREN retrieves.

## Performance Notes About the Asset Tags

- ASSET.LOAD and ASSET.CHILDREN are database queries, so you should use them only when necessary, because queries to the database take time. For example, you

might want to include error checking code after an ASSET.LOAD tag and before its subsequent ASSET.CHILDREN tag that determines whether an asset was returned by the ASSET.LOAD. If there is no asset, there is no reason to invoke the ASSET.CHILDREN tag.

- An ASSET.SCATTER call takes much longer than a single ASSET.GET call.

## Tags That Create Assetsets (Flex Assets)

Assetset tags specify a set of one or more flex assets that you want to retrieve from the database.

You can retrieve the following information from an assetset:

- The values for one attribute for each of the flex assets in the assetset
- The values for multiple attributes for each of the flex assets in the assetset
- A list of the flex assets in the assetset
- A count of the flex assets in the assetset
- A list of unique attribute values for an attribute for all flex assets in the assetset
- A count of unique attribute values for an attribute for all flex assets in the assetset

The following tables describe the assetset tags that you will use most frequently.

| ASSETSET.SETASSET (XML) | assetset:setasset (JSP) |
|---|---|
| `<ASSETSET.SETASSET`<br>`NAME="assetsetname"`<br>`TYPE="assettype"`<br>`ID="assetid"`<br>`[LOCALE="localeobject"]`<br>`[DEPTYPE="exact\|exists\|none"]`<br>`/>` | `<assetset:setasset`<br>`name="assetsetname"`<br>`type="assettype" id="assetid"`<br>`[locale="localeobject"]`<br>`[deptype="exact\|exists\|none"]/>` |

ASSETSET.SETASSET builds an asset set from a single asset that you specify and defines a compositional dependency between the template or element that it appears in and the content that it retrieves.

| ASSETSET.SETSEARCHEDASSETS (XML) | assetset:setsearchedassets (JSP) |
|---|---|
| `<ASSETSET.SETSEARCHEDASSETS`<br>`NAME="assetsetname"`<br>`[ASSETTYPES="assettype"]`<br>`[CONSTRAINT="searchstateobject"]`<br>`[LOCALE="localeobject"]`<br>`[SITE="siteidentifier"]`<br>`[DEPTYPE="exact\|exists\|none"]/>` | `<assetset:setsearchedassets`<br>`name="assetsetname"`<br>`[assettypes="assettype"]`<br>`[constraint="searchstateobject"]`<br>`locale="localeobject"]`<br>`[site="siteidentifier"]`<br>`[deptype="exact\|exists\|none"]/>` |

ASSETSET.SETSEARCHEDASSETS creates an assetset object which represents all assets of specific types narrowed by specified search criteria (represented by the searchstate object that you name in the constraint parameter).

This tag also defines a compositional dependency between the template or element in which it appears and the each asset in the set.

| ASSETSET.GETMULTIPLEVALUES (XML) | assetset:getmultiplevalues (JSP) |
|---|---|
| `<ASSETSET.GETMULTIPLEVALUES NAME="assetsetname" LIST="listname" [BYASSET="true\|false"] PREFIX="prefix"/>` | `<assetset:getmultiplevalues name="assetsetname" list="listname" [byasset="true\|false"] prefix="prefix"/>` |

`ASSETSET.GETMULTIPLEVALUES` scatters attribute values from several attributes (and potentially more than one asset) into several specified lists.

FatWire recommends using `ASSETSET.GETMULTIPLEVALUES` when the goal is to display a fixed-format table of assets, or to obtain many attributes of a single asset (such as for a product detail page).

`ASSETSET.GETMULTIPLEVALUES` has the following limitations:

- Only non-foreign attributes can be scattered.
- Text-type attributes cannot be scattered.

| ASSETSET.GETATTRIBUTEVALUES (XML) | assetset:getattributevalues (JSP) |
|---|---|
| `<ASSETSET.GETATTRIBUTEVALUES NAME="assetsetname" ATTRIBUTE="attribname" [TYPENAME="assettypename"] LISTVARNAME="varname" [ORDERING="ascending\|descending"]/>` | `<assetset:getattributevalues name="assetsetname" attribute="attribname" [typename="assettypename"] listvarname="varname" [ordering="ascending\|descending"]/>` |

`ASSETSET.GETATTRIBUTEVALUES` gets the list of values for a specified attribute of the assets represented by an assetset.

| ASSETSET.GETASSETLIST (XML) | assetset:getassetlist (JSP) |
|---|---|
| `<ASSETSET.GETASSETLIST NAME="assetsetname" [LIST="attriblist"] [MAXCOUNT="rowcount"] [METHOD="random\|highest"] LISTVARNAME="varname/>` | `<assetset:getassetlist name="assetsetname" [list="attriblist"] [maxcount="rowcount"] [method="random\|highest"] listvarname="varname"/>` |

`ASSETSET.GETASSETLIST` retrieves an ordered list of assets, given optional sort criteria. The resulting list has two columns, assetid and assettype, that are sorted by the criteria that you specify.

## Tags That Create Searchstates (Flex Assets)

Searchstate tags assemble criteria that filter the assets that you retrieve using the `assetset` tags.

You build a searchstate by adding or removing constraints to narrow or broaden the list of flex assets that are described by the searchstate.

The following tables describe the `searchstate` tags that you will use most frequently.

| SEARCHSTATE.CREATE (XML) | searchstate:create (JSP) |
|---|---|
| `<SEARCHSTATE.CREATE NAME="ssname" [OP="and\|or"]/>` | `<searchstate:create name="ssname" [op="and\|or"]/>` |

`SEARCHSTATE.CREATE` builds an empty searchstate object. You must begin constructing a searchstate with this tag.

| SEARCHSTATE.ADDSTANDARDCONSTRAINT (XML) | searchstate:addstandardconstraint (JSP) |
|---|---|
| `<SEARCHSTATE.ADDSTANDARDCONSTRAINT NAME="ssname" [BUCKET="bucketname"] [TYPENAME="assettype"] ATTRIBUTE="attribname" [LIST="listname"] [IMMEDIATEONLY="true\|false"] [CASEINSENSITIVE="true\|false"]/>` | `<searchstate:addstandardconstraint name="ssname" [bucket="bucketname"] [typename="assettype"] attribute="attribname" [list="listname"] [immediateonly="true\|false"] [caseinsensitive="true\|false"]/>` |

`SEARCHSTATE.ADDSTANDARDCONSTRAINT` adds an attribute name/value constraint into a new or existing searchstate object.

You can constrain the attribute by a list of values that you specify in the `list` parameter.

| SEARCHSTATE.ADDSIMPLESTANDARDCONSTRAINT (XML) | searchstate:addsimplestandardconstraint (JSP) |
|---|---|
| `<SEARCHSTATE.ADDSIMPLESTANDARDCONSTRAINT NAME="ssname" [BUCKET="bucketname"] [TYPENAME="assettype"] ATTRIBUTE="attribname" VALUE="value" [IMMEDIATEONLY="true\|false"]/>` | `<searchstate:addsimplestandardconstraint name="ssname" [bucket="bucketname"] [typename="assettype"] attribute="attribname" value="value" [immediateonly="value"]/>` |

`SEARCHSTATE.ADDSIMPLESTANDARDCONSTRAINT` adds an attribute name/single value constraint to an existing searchstate.

This tag is the simple version of SEARCHSTATE.ADDSTANDARDCONSTRAINT. The object referred to by NAME is updated to reflect the new constraint. If the attribute name is already in the searchstate, then the new constraint replaces the old constraint.

| SEARCHSTATE.ADDRANGECONSTRAINT (XML) | searchstate:addrangeconstraint (JSP) |
|---|---|
| `<SEARCHSTATE.ADDRANGECONSTRAINT NAME="ssname" [BUCKET="bucketname"] [TYPENAME="assettype"] ATTRIBUTE="attribname" LOWER="lowrange" UPPER="uprange" [CASEINSENSITIVE="true\|false"]/>` | `<searchstate:addrangeconstraint name="ssname" [bucket="bucketname"] [typename="assettype"] attribute="attribname" lower="lowrange" upper="uprange" [caseinsensitive="true\|false"] />` |

`SEARCHSTATE.ADDRANGECONSTRAINT` adds a range constraint for a specific attribute name.

| SEARCHSTATE.ADDRICHTEXTCONSTRAINT (XML) | searchstate:addrichtextconstraint (JSP) |
|---|---|
| `<SEARCHSTATE.ADDRICHTEXTCONSTRAINT NAME="ssname" [BUCKET="bucketname"] [TYPENAME="assettype"] ATTRIBUTE="attribname" VALUE="criteria" [PARSER="parsername"] CONFIDENCE="minlevel" [MAXCOUNT="number"] />` | `<searchstate:addrangeconstraint name="ssname" [bucket="bucketname"] [typename="assettype"] attribute="attribname" lower="lowrange" upper="uprange" [caseinsensitive="true\|false"] />` |

`SEARCHSTATE.ADDRICHTEXTCONSTRAINT` adds an attribute name and rich-text expression to the list of rich-text constraints in the searchstate.

| SEARCHSTATE.TOSTRING (XML) | searchstate:tostring (JSP) |
|---|---|
| `<SEARCHSTATE.TOSTRING NAME="objname" VARNAME="varname"/>` | `<searchstate:tostring name="objname" varname="varname"/>` |

SEARCHSTATE.TOSTRING converts a searchstate object into its string representation that is suitable for various uses, such as saving in a session variable or packing into a URL.

| SEARCHSTATE.FROMSTRING (XML) | searchstate:fromstring (JSP) |
|---|---|
| `<SEARCHSTATE.FROMSTRING NAME="objname" VALUE="stringval"/>` | `<searchstate:fromstring name="objname" value="stringval"/>` |

`SEARCHSTATE.FROMSTRING` provides the ability for a searchstate object to be initialized from its string representation. You must create an empty searchstate using the `SEARCHSTATE.CREATE` tag before you can use this tag.

# Variables

Content Server supports the following kinds of variables:

- Regular variables, which last for the duration of the current template or element, unless you explicitly remove them. Regular variables have a global scope.
- Session variables, which last for the duration of the current session.

Content Server provides several standard variables whose names are reserved. You can retrieve the values of these variables, but you cannot use their names for other variables that you create.

This section describes the following topics:

- Reserved Variables
- Setting Regular Variables
- Setting Session Variables
- Working With Variables
- Variables and Precedence
- Best Practices with Variables

## Reserved Variables

The following table defines the standard Content Server variables. Unless otherwise noted, these are regular variables:

| Variable | Definition |
|---|---|
| `tablename` | A variable that is set to a tablename before the `execsql` tags can be run. |
| `pagename` | The name of the Content Server page being invoked. |
| `ftcmd` | A variable used in calls to CatalogManager. |
| `username` | A session variable that contains the name of the user who is currently logged in to the current session. |

| Variable | Definition |
|---|---|
| `password` | A session variable that contains the password of the user who is currently logged in to the current session. |
| `authusername` | A variable that you can set to the username of a user who you want to log in to Content Server. This can be sent to Content Server via a URL. |
| `authpassword` | A variable that you can set to the password of a user who you want to log in to Content Server. This can be sent to Content Server via a URL. |
| `currentACL` | A session variable that contains the ACLs that the current user belongs to. |
| `errno` | Error numbers reported by Content Server tags. |
| `context` | Reserved for future use in the `render:calltemplate` tag. For more information, see the *Developer's Tag Reference*. |
| `site` | The full name of the site, as stored in the `name` column of the `Publication` table.<br>The `site` variable is set as a `resarg` in all of the Template and Site Entry assets. The site owns the Template and SiteEntry assets that you create within the site. |
| `sitepfx` | The site prefix (and short name of the site), as stored in the `cs_prefix` column of the `Publication` table. |
| `ft_ss` | An internal variable that is automatically set by Content Server to support communication with Satellite Server. When `ft_ss` is set to `true`, Content Server infers that a request is from Satellite Server. |
| `c` | The asset type that a template formats. CS-Direct sets this variable by default when you save the Template asset. |
| `cid` | The ID of the asset being rendered or formatted by a template. |
| `ct` | The value of a child template, if there is one. For a thorough explanation of child templates, see "Example 3: Using the ct Variable" on page 627. |
| `p` | The ID of an asset's parent page, if there is one. |
| `rendermode` | Specifies whether a page entry is to be delivered live, exported, or previewed. By default, `rendermode` is `live`. When you use Export to Disk, or the Preview function, CS-Direct automatically overrides the value of this variable with `export` or `preview`. This value is used internally and must not be modified. |
| `seid` | The ID of a SiteEntry asset. |
| `tid` | The ID of a Template asset. |
| `eid` | The ID of a CSElement asset, `eid` is available to the CSElement's root element. |

## Setting Regular Variables

Most of the variables that you will use while coding Content Server templates and elements are **regular variables**. Regular variables last for the duration of the current template or element, unless they are explicitly deleted using Content Server tags.

### Setting Variables with SETVAR

Inside a Content Server element, you can call the `SETVAR` XML or JSP tags to create a variable and establish its initial value. For example, the following `SETVAR` XML tag creates a variable named `dog` and sets its value to `fido`:

```
<SETVAR NAME="dog" VALUE="fido"/>
```

If the variable already exists, `SETVAR` resets its value to the new value. For example, the following command resets the value of dog to `mocha`:

```
<SETVAR NAME="dog" VALUE="mocha"/>
```

### Setting Variables via a URL

Content Server creates a page when a browser goes to a URL managed by a Content Server application. Each page is associated with a particular URL. Imagine, for example, a page associated with a URL having the following format:

```
http://host:port/servlet/ContentServer?pagename=Experiment/
Hello
```

At the end of every URL, you can set one or more variables. For example, the following URL creates three variables in the `Hello` page:

```
http://host:port/servlet/ContentServer?pagename=Experiment/
Hello&dog=fido&cat=fifi
```

The preceding URL creates the following variables available to `Hello`:

- A variable named `pagename` whose value is `Experiment/Hello`
- A variable named `dog` whose initial value is `fido`.
- A variable named `cat` whose initial value is `fifi`.

### Setting Default Variables for Elements and Templates with Content Server Explorer

You can use Content Server Explorer to create default variables in a page by placing the variables in either of the following fields:

- `resargs1` or `resargs2` fields of the `SiteCatalog` database table
- `resdetails1` or `resdetails2` fields of the `ElementCatalog` database table

For example, you can use Content Server Explorer to access the `SiteCatalog` table, and then create variables `dog` and `cat` by placing name/value pairs in the `resargs1` and `resargs2` fields:

| pagename | rootelement | csstatus | resargs1 | resargs2 | cacheinfo | acl |
|---|---|---|---|---|---|---|
| Hello | Experiment/Hello | Live | dog=fido | cat=fifi | | |

Note that we placed one name/value pair in `resargs1` and another in `resargs2`. Alternatively, we could have put both name/value pairs in `resargs1`, as shown in the following diagram:

| pagename | rootelement | csstatus | resargs1 | resargs2 | cacheinfo | acl |
|---|---|---|---|---|---|---|
| * Hello | Experiment/Hello | Live | dog=fido& cat=fifi | | | |

You can also set the values of `dog` and `cat` in the `ElementCatalog` table by putting name/value pairs in the `resdetails1` and `resdetails2` fields:

| elementname | description | url | resdetails1 | resdetails2 |
|---|---|---|---|---|
| • Hello | | Experiment\Hello.xml | dog=fido | cat=fifi |

Variables set through the URL or through `POST` and `GET` operations take precedence over variables set using the `SiteCatalog` or `ElementCatalog` tables. For example, if a URL sets variable `dog` to `rex` and the `SiteCatalog` sets `dog` to `fido`, then the resulting value of `dog` will be `rex`.

### Setting Variables Using HTML Forms

In CGI programming, a buyer fills out a form. Then, the browser encodes the buyer's responses as name/value pairs, which get passed to the CGI script.

Although Content Server does not use traditional CGI programming, a Content Server element can still display a form. As in traditional programming, the browser encodes the buyer's responses as name/value pairs. However, instead of passing these name/value pairs to a CGI program, the pairs get passed to a different Content Server page. The receiving Content Server page can access the name/value pairs as it would access any Content Server variable.

Cookie names and values are also instantiated as variables. For more information about cookies, see Chapter 9, "Sessions and Cookies."

## Setting Session Variables

HTTP is a stateless protocol. To overcome this limitation, Content Server can maintain state between requests and, thus, keep track of sessions.

A browser connection to the Content Server system establishes a session. Thereafter, the session is uniquely identified to the system. Content Server can deliver pages whose content and behavior are based on this unique identity.

When a client first enters your site, a unique session is established. Content Server associates a default user identity with a new session and maintains that information in session variables. **Session variables** contain values that are available for the duration of the session. They are saved as part of the user's session and are used to retain the value of a variable across page requests.

In a clustered configuration, the session state is maintained across all cluster members. Session variables should be used carefully, since there is a resource cost that is proportionate to the number and size of session variables used.

Session state is lost under these conditions:

- The client exits.

- The session has timed out. Content Server can optionally terminate a session if no requests have been made for some period of time.
- The application server has been restarted.

Server resources associated with the session are deallocated when the following occurs:

- The session has been explicitly terminated by the client via a Content Server tag.
- The session has timed out.
- The application server has been restarted.

Use the `SETSSVAR` XML and JSP tags to create a session variable. If the session variable already exists, `SETSSVAR` resets the variable's value. For example, the following `SETSSVAR XML` tag sets the session variable `profile` to the value `10154`:

```
<SETSSVAR NAME="profile" VALUE="10154"/>
```

# Working With Variables

The following sections describe how to work with Content Server variables.

## Retrieving a Variable's Value

The syntax you use to read the value of a variable depends on the kind of variable:

| Type of Variable | Syntax | Example |
|---|---|---|
| String Variable | `Variables.`*`variable_name`* | `Variables.dog` |
| Counter Variable | `Counters.`*`variable_name`* | `Counters.position` |
| Session Variables | `SessionVariables.`*`variable_name`* | `SessionVariables.username` |
| Property | `CS.Property.`*`property_name`* | `cs.use.short.jsp.names` |

Content Server XML provides quite a few methods for accessing list variables.

## Displaying a Variable's Value

Use the `CSVAR` XML tag to display the value of any kind of variable, including properties and session variables. Use the `ics:getvar` JSP tag to view the value of a regular Content Server variable; or the `ics:getssvar` JSP tag to display the value of a session variable. For example, if the following code appears in an XML element:

```
<SETVAR NAME="mood" VALUE="happy"/>
<p>My dog is <CSVAR NAME="Variables.mood"/>.</p>
```

then the resulting page displays the following text:

```
My dog is happy.
```

You can also include literal values as part of the `NAME` argument to the `CSVAR` XML tag; for example, the following code will also generate "`My dog is happy.`", but evaluates more slowly:

```
<SETVAR NAME="mood" VALUE="happy"/>
<p><CSVAR NAME="My dog is Variables.mood"/>.</p>
```

## Assigning One Variable Value to Another Variable

You can assign the value of one variable to another variable. You accomplish this task differently if you are coding with XML than if you are coding with JSP.

### JSP

If you are coding with JSP, you cannot use the `ics:getvar` tag to evaluate the variable value because you cannot nest one JSP tag within another JSP tag. To circumvent this limitation, use the `ics.GetVar` Java method to substitute variable values, as shown in the following sample code:

```
<ics:setvar name="myVar" value="Fred"/>
<ics:setvar name="yourVar" value='<%=ics.GetVar("myVar")%>'/>
<ics:getvar name="yourVar"/>
```

**Note**

You must enclose the expression that evaluates the variable value (`'<%=icsGetVar("myVar")%>'` in the example) in single quotes. Otherwise your JSP element will throw an exception.

### XML

The following lines of XML assign the value `carambola` to a variable named `your_favorite`:

```
<SETVAR NAME="my_favorite" VALUE="carambola"/>
<SETVAR NAME="your_favorite" VALUE="Variables.my_favorite"/>
```

Taking this one step further, you can concatenate two variable values and assign the result to a third variable. For example, the following sets the variable `car` to the value `red rabbit`.

```
<SETVAR NAME="color" VALUE="red"/>
<SETVAR NAME="model" VALUE="rabbit"/>
<SETVAR NAME="car" VALUE="Variables.color Variables.model"/>
```

## Using Variables in HTML Tags

You can use XML and JSP variables within traditional HTML tags, although you code differently to accomplish this in XML and JSP.

### JSP

If you are coding with JSP, you use the `ics:getvar` tag or the `ics.GetVar` Java method to evaluate the variable value.

You can also use the `ics.resolvevariables` tag to resolve variables that are contained within a string. For example, the following code displays the phrase "The date is," along with the value of the `CS.Date` variable:

```
<ics.resolvevariables name="The date is $(CS.Date)."
delimited="true"/>
```

The `delimited` parameter indicates that you have used the delimiters `$(` and `)` to explicitly mark the variable or variables that you want to resolve. If you want to use variables to specify a list name and a column in that list, for example, you use the following syntax:

```
<ics.resolvevariables
name="$(Variables.listname).$(Variables.columnname)"
delimited="true"/>
```

If the `delimited` parameter is set to `false`, no delimiters are used to set off variables.

### XML

You can use XML variables inside HTML tags if you use the appropriate attributes. For example, the following code does not contain the appropriate attributes and, therefore, does not set the background color to red:

```
<SETVAR NAME="color" VALUE="red"/>
<TABLE bgcolor="Variables.color">
...
```

To use XML variable values within an HTML tag, you must use the REPLACEALL attribute within that HTML tag. The REPLACEALL attribute tells the system to substitute the current value of this XML variable within this HTML tag. Therefore, the correct way to code the preceding lines is as follows:

```
<SETVAR NAME="color" VALUE="red"/>
<TABLE bgcolor="Variables.color" REPLACEALL="Variables.color">
...
```

You can combine multiple variable values within one REPLACEALL attribute. For example, the following HTML TABLE tag uses two XML variables:

```
<SETVAR NAME="color" VALUE="red"/>
<SETVAR NAME=”myborder” VALUE=”3”/>
<TABLE bgcolor="Variables.color" border=”Variables.myborder”
     REPLACEALL="Variables.color,Variables.myborder">
...
```

The <REPLACEALL> tag is an alternative to the REPLACEALL attribute. The <REPLACEALL> tag performs substitutions within its domain; for example:

```
<SETVAR NAME="highlight" VALUE="red"/>
<SETVAR NAME="diminish"  VALUE="gray"/>
<REPLACEALL LIST="Variables.highlight,Variables.diminish">
<TABLE>
  <TR BGCOLOR="Variables.highlight"><TD>Diamonds</TD></TR>
  <TR BGCOLOR="Variables.highlight"><TD>Pearls</TD></TR>
  <TR><TD>Malachite</TD></TR>
  <TR BGCOLOR="Variables.diminish"><TD>Coal</TD></TR>
</TABLE>
</REPLACEALL>
```

The output of this section is:

The REPLACEALL tag performs a string search and replace, and is, therefore, potentially very slow. Use the REPLACEALL attribute where possible. If you must use the REPLACEALL tag, keep the amount of code you enclose with it as small as possible.

## Evaluating Variables with IF/THEN/ELSE

Content Server XML and JSP provides the IF/THEN/ELSE construct available in most computer languages. However, the only conditional operation for variables is to compare two values for equality or inequality. You can’t, for example, compare two values to see if one is greater than another. (You can write Java code to do that, however.)

For example, the following code branches depending on the value of a variable named greeting.

```
<IF COND="Variables.greeting=Hello">
<THEN>
     <p>Welcome.</p>
```

```
</THEN>
<ELSE>
     <p>So long.</p>
</ELSE>
</IF>
```

If greeting is set to `Hello`, then Content Server generates the HTML:

```
<p>Welcome.</p>
```

If greeting is set to anything other than `Hello`, Content Server generates:

```
<p>So long.</p>
```

## Variables and Precedence

Variables set through a URL or through HTTP GET and POST operations take precedence over variables set with the `resargs` and `resdetails` columns in the `SiteCatalog` and `ElementCatalog` tables.

## Best Practices with Variables

Because all variables are global and the syntax for accessing variables from items in lists and from other sources is the same, good coding practices help you to avoid errors. For example:

- Because it is easy to reuse base names in your elements, use prefixes in front of variables to define them uniquely. The recommended syntax to use is: `Variables.`*`assettype`*`:`*`fieldname`*.

  For example, `Variables.Article:description`.

  The `ASSET.SCATTER` tag makes it easy for you to use this syntax through its `PREFIX` attribute. (For more information about this tag, see Chapter 27, "Coding Elements for Templates and CSElements.")
- If you are going to use the `RESOLVEVARIABLES` tags to resolve your variables, set the `DELIMITED` parameter to `true` and use the delimiters `$(` and `)` to explicitly indicate the variables you want to resolve.
- Use debugging to catch naming conflicts. Use the Property Editor to set the `com.fatwire.logging.cs` property (in the `commons-logging.properties` file or in `log4j.properties`, depending on which logging framework you are using). When this property is enabled, Content Server writes a record of all the variables that are created to the `futuretense.txt` log.

For a list of the error values that Content Server and CS-Direct tags can write to the `errno` variable, see the *Content Server Tag Reference*.

# Other Content Server Storage Constructs

In addition to regular and session variables, Content Server supports a number of storage constructs. The following sections describe these constructs and how to use them:

- Built-ins
- Lists
- Counters

## Built-ins

Content Server provides several built-ins, which return values such as the current date.

The general syntax of a built-in is:

```
CS.builtin
```

For example, `UniqueID` is a built-in that generates a unique ID. The following syntax generates or references this built-in variable:

```
CS.UniqueID
```

For a list of built-ins in Content Server, see the *Content Server Tag Reference*.

## Lists

A list consists of a table of values organized in rows and columns. Use the `SETROW` or `GOTOROW` tags to identity the proper row.

The following entities create lists:

- The `SELECTTO`, `EXECSQL`, `CATALOGDEF`, `STRINGLIST` and `CALLSQL` tags
- CatalogManager commands
- TreeManager commands
- Custom tags

Use the following syntax to refer to a current row's column value:

```
listname.colname
```

For example, if a list named `cars` had a column named `color`, the value of the current row would be referenced as:

```
cars.color
```

### Looping Through Lists

Use the `LOOP` XML tag or the `ics:listloop` JSP tag to iterate through a list. For each row in the list, Content Server executes the instructions between the loop tags.

For example, consider a table named `MyCars` containing the following rows:

| id | Model | Color | Year |
|---|---|---|---|
| 224 | Ford Focus | blue | 2001 |
| 358 | VW Rabbit | red | 1998 |

| id | Model | Color | Year |
|---|---|---|---|
| 359 | Toyota Corolla | yellow | 2000 |
| 372 | Alpha Romeo Spider | red | 1982 |
| 401 | Porsche 911 | red | 1984 |
| 423 | Dodge Voyager | tan | 1991 |

The following XML searches `MyCars` for red cars. The `SELECTTO` XML and JSP tags write this information into a list variable named `carlist`.

```
<SETVAR NAME="color" VALUE="red"/>
<SELECTTO FROM="MyCars" WHERE="color" WHAT="*" LIST="carlist"/>
Red cars: <BR/>
<OL>
<LOOP LIST="carlist">
    <LI><CSVAR NAME=”carlist.model"/> </LI>
</LOOP>
</OL>
```

The preceding XML generates the following HTML:

```
Red cars: <BR/>
<OL>
 <LI> VW Rabbit </LI>
 <LI> Alpha Romeo Spider </LI>
 <LI> Porsche 911 </LI>
</OL>
```

## Counters

A counter is an XML variable whose value is an integer. Three tags control counters:

| Tag | What It Does |
|---|---|
| SETCOUNTER | Initializes a counter variable |
| INCCOUNTER | Changes the counter’s value by a specified amount |
| REMOVECOUNTER | Destroys the counter variable |

To create a counter, you call `SETCOUNTER`. To change its value, call `INCCOUNTER`. For example, consider the following code:

```
<SETCOUNTER NAME="c" VALUE="10"/>
<INCCOUNTER NAME="c" VALUE="3"/>
<p>Current value is <CSVAR NAME="Counters.c"/></p>
```

The output of this code is:

```
Current value is 13
```

Notice that you reference counter variables using the syntax:

`Counters.`*`name`*

# Values for Special Characters

If you need to use special (non-alphanumeric) characters in your XML or JSP, you will need to use their hexadecimal character representation. For example, the following line specifies a space as part of a variable value:

```
<SETVAR NAME="foo" VALUE="foo%20bar"/>
```

The following are hexadecimal values for special characters that are commonly used in Content Server:

| Hexadecimal Value | Character |
|---|---|
| %22 | doublequote (") |
| %20 | one space |
| %3c | less than sign (<) |
| %3e | greater than sign (>) |
| %26 | ampersand (&) |
| %09 | tab (\t) |
| %0a | newline (\n) |
| %0d | carriage return (\r) |
| %25 | percent (%) |

Chapter 5

# Page Design and Caching

Caching your web pages can improve your site's performance. Whether your site is static or dynamic, you need to design your site so that part or all of a given page is cached. This chapter describes how Content Server caching works, and includes the following sections:


- Modular Page Design
- Caching
- Viewing the Contents of the Satellite Server Cache
- Double-Buffered Caching

# Modular Page Design

FatWire recommends that you design your web pages using a modular page design strategy, where a web page that a web site visitor sees is composed of multiple elements. Modular page design has several benefits:

- It improves system performance by allowing you to develop an efficient caching strategy
- It allows you to code common design elements, like navigation bars, one time and use them on multiple web pages

The following diagram shows a simple modular page:

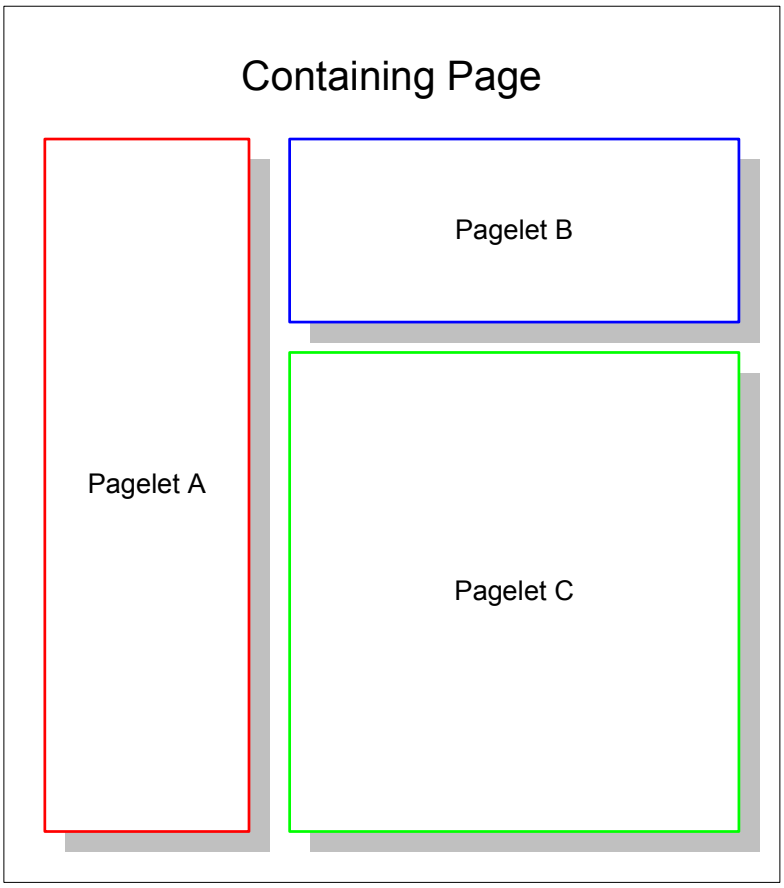



**Figure 3:** A modular page

Each rectangle in Figure 5 represents a pagelet—the generated output of one or more elements. These pagelets are called by a containing page. The containing page lays out how the pagelets appear on the finished page and contains any code that must be evaluated each time the page is viewed—custom ACL checking code, for example. This strategy allows you to code an element once and use it in many places in your web site.

# Caching

Content Server allows you to cache entire web pages and/or the components that make up those web pages. An efficient page caching strategy improves system performance by reducing load.

Two members of the Content Server product family implement page caching:

- Content Server, which caches pages on the Content Server system
- Satellite Server, which provides a second level of caching for your Content Server system, and can also be used as a remote cache for your web pages

Content Server utilizes both the Content Server and Satellite Server caches to create an efficient caching strategy.

## Content Server Caching

Pagelets generated by requests to the ContentServer servlet can be cached on disk. If a page is accessed frequently and its content depends on a small number of parameters, then it is a good candidate for disk caching.

To disk-cache a pagelet, you use one of the following tags:

| Product | JSP Tag | XML Tag |
|---|---|---|
| Content Server | `satellite:page` | `SATELLITE.PAGE` |
| Content Server Direct | `render:satellitepage` | `RENDER.SATELLITEPAGE` |

If the pagelet that you want to cache is not already in the disk cache, ContentServer adds it to the cache and then serves the pagelet. If the specified pagelet is already in the disk cache, ContentServer simply serves it.

The expiration of disk-cached pagelets is time-based and governed by properties in the `futuretense.ini` file, in conjunction with the values set in the `cscacheinfo` and `sscacheinfo` columns of the `SiteCatalog` table (`cscacheinfo` corresponds to the Content Server servlet; `sscacheinfo` corresponds to the Satellite Server servlet). When a cached page is requested, its expiration time (cache timeout) is stored in the `etime` column of the `SystemPageCache` table. The expiration time for cached pages requested through the Content Server servlet is calculated from the `cscachinfo` value, and the expiration time for cached pages requested through the Satellite Server servlet is calculated from the `sscacheinfo` value. Items in cache are bound by the same security rules as uncached pages; Content Server ACLs apply to cached pagelets just as they do to elements.

## BlobServer and Caching

The term **blob** is an acronym for **b**inary **l**arge **ob**ject. Although a blob is usually an image file, a blob can be any binary object, including a Microsoft Word file or a spreadsheet. Most web sites serve a number of blobs.

To serve blobs, Content Server offers a special servlet called BlobServer. The BlobServer gathers a blob from a table and performs all relevant security checks.

You access BlobServer with the BlobServer tags:

- For Content Server systems: `satellite:blob`
- For CS-Direct systems: `render:satelliteblob`

Both of these tags cache blobs in the Content Server and Satellite Server caches. For more information about the BlobServer tags, see the *Content Server Tag Reference*.

### Deleting Blobs from the Content Server Memory Cache

To delete a specific blob from the Content Server cache, you must do the following in the BlobServer URL:

- Rename the `blobtable` parameter to `flushblobtable`
- Authenticate as a user with `SiteGod` privileges by passing credentials through the `authusername` and `password` parameters

For example:

```
http://hostname:port/servlet/
BlobServer?blobcol=urlpicture&blobheader=image%2Fgif&blobkey=id
&flushblobtable=NewPortalImage&blobwhere=22&authusername=username&
authpassword=password
```

To delete **all** blobs, rename the `blobtable` parameter to **`flushblobtables`** (notice the “s”) and set its value to `true`.

## Satellite Server Caching

Satellite Server, automatically installed with Content Server, provides an additional layer of caching. To improve your Content Server system’s performance, you can add remote Satellite Server systems, putting your content closer to its intended audience.

Satellite Server caches pages, pagelets, and blobs to disk or to memory. You can use the `Inventory` servlet to view the contents of the memory and disk caches in varying degrees of detail. Note that items cached on Satellite Server are not protected by Content Server APIs. You can overcome this limitation by using the caching strategy outlined in “Pagelet Caching Strategies” on page 132.

Satellite Server caches small items to memory and large items to disk. You control the definitions of small and large through the `file_size` property. For more information on setting Satellite Server properties, see the *Content Server Property Files Reference*.

On a busy site, each Satellite Server system’s cache fills up quickly with the most popular pages. When the cache is full, Satellite Server deletes old pages to make room for new ones. Satellite Server uses a Least Recently Used algorithm (LRU) to determine which items should be removed from the cache. In other words, when a new page needs to be cached, Satellite Server removes the page that hasn’t been accessed for the longest time. For example, given two cached pages—one that hasn’t been accessed in 36 hours and the other that hasn’t been accessed in 2 hours—Satellite Server removes the page that hasn’t been accessed in 36 hours.

### Cache Expiration

Page and pagelet expiration on Satellite Server is specified in the `sscacheinfo` column of the `SiteCatalog` table. Each time a page or pagelet is invoked through Satellite Server, Satellite Server processes the `sscacheinfo` field's value and determines when the

page or pagelet should expire. Consult "CacheInfo String Syntax" on page 148 for information about the `sscacheinfo` field.

**Note**

Deprecation notice: It is possible to override the `sscachinfo` expiration information for pagelets by specifying the `cachecontrol` attribute in the `satellite.page` and `render.satellitepage` tags. However, this practice is deprecated because it can lead to non-deterministic behavior: some pagelets may be accessed through the default method—without the `cachecontrol` attribute—while others may be accessed with an override. The first method invoked will set the expiration for Satellite Server, and the second one will have no effect on the expiration.

Blobs cached on Satellite Server expire using the following algorithm:

- You can use Satellite Server tags to override the default expiration time on a blob-by-blob basis.
- If there is no Satellite tag to override the default expiration, Satellite Server gets the expiration time from the value of the `satellite.blob.cachecontrol.default` property. This property is described in "Content Server Page Caching Properties" on page 127.
- If no value is set for the `satellite.blob.cachecontrol.default` property, Satellite Server gets the expiration time from the value of the `expiration` property, described in "Satellite Server Properties" on page 128.

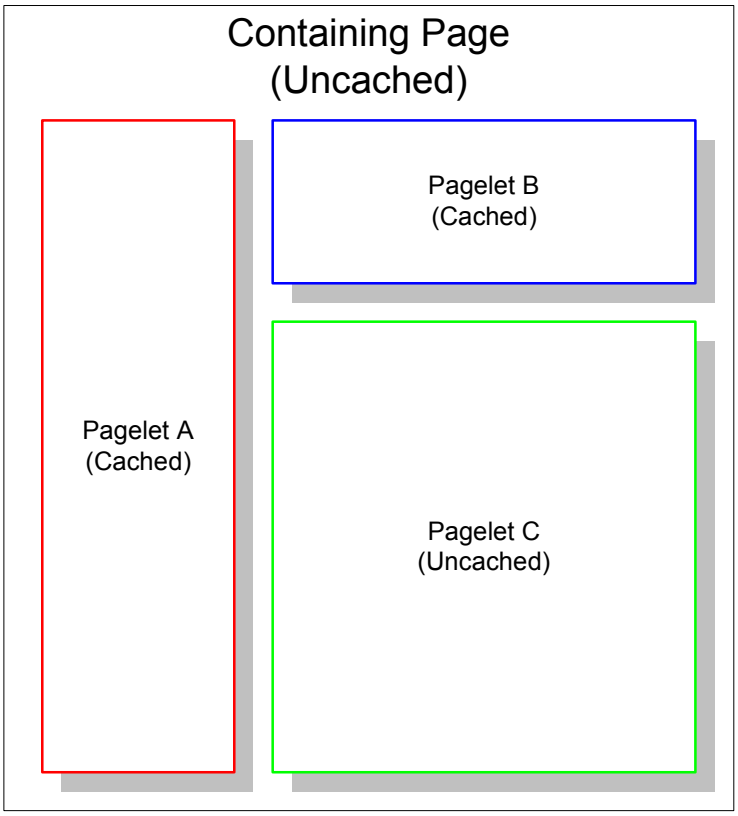
## Caching with the Satellite Servlet

The following sections describe how the Satellite servlet caches web pages and how you can implement Satellite Server caching on your site. Use caching with the Satellite servlet in tandem with modular page design to create a fast, efficient web site.

### How the Satellite Servlet Caches Pages

The Satellite servlet allows caching at the pagelet level. To implement caching with the Satellite servlet, you use Satellite Server XML or JSP tags in your Content Server pages, and you access pages using special Satellite URLs.

For example, suppose that you used the Satellite servlet to implement pagelet-level caching on a web page named `myPage`. `myPage`, shown in Figure 3, is composed of a containing page and three pagelets: A, B, and C. The containing page and pagelets A and B are already cached on a Satellite Server system, but pagelet C is not cached.




When a user requests `myPage`:

1. Satellite Server examines the URL. If it is a Satellite URL, the Satellite servlet gets the cached copy of the containing page. The servlet then looks for pointers to pagelets which are not currently in its cache, and requests those pagelets from Content Server. So, in our example, the Satellite servlet gets the containing page, and gets pagelets A and B from its cache.
2. The Satellite servlet requests Pagelet C from Content Server.
3. Content Server parses the appropriate XML to create Pagelet C and sends it to the Satellite servlet.
4. The Satellite servlet assembles Pagelets A, B, and C into the page, and sends the assembled page to the requester. The servlet also caches Pagelet C.

### Implementing Caching with the Satellite Servlet

To implement pagelet-level caching with the Satellite servlet, you add Satellite tags to your Content Server templates. You do not develop any XML, JSP, or Java code on Satellite Server systems. In fact, Satellite Server does not know how to parse XML.

The Satellite tags in your elements are interpreted by the Java code you installed as part of Satellite Server. If this code is being called with a Satellite URL, it generates the information that the Satellite servlet uses to cache and construct the pagelets. If you do not call an element containing Satellite tags with a Satellite URL, the resulting page functions as if the Satellite tags were Content Server tags.

Satellite URLs look like the following example:

`http:\\host_name:port/servlet/Satellite?pagename=page`

where *host_name* and *port* are the hostname and port number of your Satellite Server machine, and `page` is the name of the page you are requesting. A Satellite URL can also include name/value pairs you want to pass to the called page.

### Caching a Pagelet

The following sample code uses the `render:satellitepage` tag to call a pagelet. If the pagelet is not already in Satellite Server's cache, the Satellite servlet loads and caches the page. If the pagelet encounters an error during the processing and cannot be evaluated, it is not cached.

The `render:satellitepage` tag (and the `satellite:page` tag and their xml equivalents) identifies a cached pagelet by the pagename and name/value pairs passed to it. If the parameters or the name/value pairs differ from one invocation to another, a different pagelet will be cached, even if the content generated is the same. It is important to use name/value pairs to pass arguments to a pagelet through these tags.

Values passed through the ICS object pool, ICS List pool, page attribute context, and session (including session variables) may not be available to all called pagelets, because nested pagelets may not always be called at the same time as the parent. Furthermore, pagelets that rely on session or context data are rarely cacheable anyway, so attempting to cache them can result in non-deterministic behavior.

All parameters passed to a nested pagelet through `render:satellitepage` (and the `satellite:page` tag, and their xml equivalents) must be specified in the SiteCatalog as page criteria. This is Content Server's way of determining which parameters are relevant when building a pagelet for caching. Parameters other than those listed in the SiteCatalog are not permitted (an error indicating this will be written to the log).

```
<cs:ftcs>
<html>
<body>
<render:satellitepage pagename="My/Sample/Page" />
</body>
</html>
</cs:ftcs>
```

### Caching a Blob

Using Satellite tags to load and cache a blob is similar to the way you use Satellite tags to load and cache a pagelet. The following sample code adds to the previous example by calling a blob as well as a pagelet.

Line 8 uses the `ics:selectto` tag to perform a simple SQL query that retrieves a blob from the database. Results are returned in the form of an `IList` named `imagelist`.

Line 13 uses the `satellite:blob` tag to load the blob that was retrieved from the database in line 8. As with the `satellite.page` tag, if the blob is not in Satellite's cache, Satellite will load and cache the blob. The `cachecontrol` parameter is set so that the blob will expire at a given time; in this case, every 30 minutes.

```
1   <html>
2   <body>
3   <!-- NOTE: This will fail if list has no content (== null)
    -->
```

```
4
5  <ics:setvar name="category" value="logo"/>
6  <ics:setvar name="errno" VALUE="0"/>
7  <ics:selectto from="SmokeImage" list="imagelist"
   where="category" limit="1"/>
8  <ics:if cond="IsError.Variables.errno=false">
9  <ics:then>
10 <!-- Test a blob -->
11
12 <render:satelliteblob service="img src"
   blobtable="SmokeImage"
   blobkey="id"
   blobwhere="imagelist.id"
   blobcol="urlpicture"
   blobheader="image/gif"
   cachecontrol="*:30:0 */*/*"
   alt="imagelist.alttext"
   border="0" />
13 </ics:then>
14
15 <render:satellitepage pagename="QA/Satellite/Functional/
   xml/”pagelet1"cachecontrol="never"/>
16 </body>
17 </html>
```

### Never-Expiring Blobs

If there are binary files (or blobs) on your site that seldom change or never change, such as company logos, and you are using the Satellite servlet to cache at the pagelet level, you can improve performance by using an alternative method to serve these blobs.

#### To serve never-expiring blobs

1. Copy the never-expiring images to all your Satellite Server hosts. Place them under the doc root for your web server.
2. Access the images through `<img src=”pathname”>` HTML tags rather than through `satellite:blob` Satellite tags.

For example, consider a never-expiring corporate logo file named `CorporateLogo.gif`. To use the alternative method of serving blobs, you would first copy the file to the web server’s doc root on all your Satellite Server hosts. Then, instead of serving this logo through a `satellite.blob` tag, your element could simply use a tag like the following:

```
<img src=”CorporateLogo.gif”>
```

**Note**

Be careful when using this mechanism for serving never-expiring images. For example, Satellite Server cannot warn you that one of the Satellite Server hosts does not contain the same image file as the other hosts.

```
http://myloadbalancer:1234/servlet/
ContentServer?pagename=myPage
```

The expiration of the page is controlled by the `expiration` property. For more information on the `expiration` property, see the *Content Server Property Files Reference*.

# Viewing the Contents of the Satellite Server Cache

The Inventory servlet allows you to view the various items stored in the cache. You invoke the Inventory servlet by using the following URL:

```
http://host:port/servlet/
Inventory?username=username&password=passwordword&detail
=value
```

where:

| Parameter | Description |
|---|---|
| *host:port* (required) | The host name and port number of the Satellite Server host whose cache you want to view. |
| `username` (required) | The user name that you enter to log you in to the Satellite Server host. |
| `password` (required) | The password that you enter to log you in to the Satellite Server host. |
| `detail` (optional) | The type of information you wish the Inventory servlet to display. Valid values are:<br>• `names` - Displays the header information, plus the page names of the pages in the cache.<br>• `keys` - Displays the header information, plus the page names and keys of the items in the cache.<br>If you do not supply the `detail` parameter, or if you set its value to be anything other than `name` or `keys`, the header information displays. |

The header contains the following information:

| Information type | Description |
|---|---|
| Remote host | The host that this Satellite Server system forwards requests to. |
| Maximum cache objects | The maximum number of items allowed in the cache. |
| Current size | The number of items currently in the cache. |
| Cache check interval | How often the cache is checked for expired items, in minutes. |
| Default cache expiration | The value of the `expiration` property. |

| Information type | Description |
|---|---|
| Minimum file size (in bytes) | Items larger than this value are stored in files. Items smaller than this value are stored in RAM. |

## CacheManager

Content Server's CacheManager object maintains both the Content Server and Satellite Server caches. CacheManager can do the following:

- Log pagelets in the cache tracking tables.
- Keep a record of the content (assets) that pages and pagelets contain by recording **cache dependency items** in cache-tracking tables. Cache dependency items are items that, when changed, invalidate the cached pages and pagelets that contain them. A cache dependency item is logged as a dependency for the current page and all of that page's parent pages.
- Remove pages and pagelets containing invalid items from the Content Server and Satellite Server caches.
- Rebuild the Content Server and Satellite Server caches with updated pages and pagelets after the invalid pages have been removed.

For web sites that use CS-Direct, CacheManager completes these operations automatically, ensuring that the pages are always up to date for web site visitors.

For web sites that use Content Server alone, CacheManager's cache tracking and flushing are not automatic; however you can use CacheManager's Java API to implement similar functionality on your site.

## The SiteCatalog Table

Content Server's `SiteCatalog` table lists the pages and pagelets generated by Content Server. An element must have an entry in the `SiteCatalog` table to be cached on Content Server and Satellite Server.

The fields in the `SiteCatalog` table set the default behavior of a Content Server page, including default caching behavior. For more information on the `SiteCatalog` table and its fields, see "Creating Template Assets" on page 501 and "Creating SiteEntry Assets" on page 531.

## The Cache Key

Items stored in the Content Server and Satellite Server caches are given a name called a **cache key**. The cache key uniquely identifies each item in the cache. CacheManager locates items in the cache using the cache key. Content Server and Satellite Server generate cache keys automatically, based on the values in the `pagename`, `resargs`, and `pagecriteria` fields of the `SiteCatalog` table, and other internal data.

### pagecriteria and the Cache Key

You include variables used by the page in the cache key by specifying them in a comma-separated list in the `pagecriteria` field of the `SiteCatalog` table. For example, suppose that you have a page called `myPage` which uses the values "`red`" and "`blue`." To include "`red`" and "`blue`" in `myPage's` cache key, enter

`favoritecolor,second_favoritecolor` in the `pagecriteria` column and `favoritecolor=red&second_favoritecolor=blue` in the `resargsl` column.

Content Server and Satellite Server use the `pagecriteria` and parameters that are passed to cached pages to help generate the cache keys. If the parameters differ from one invocation to another, a different page will be cached even if the content being generated is the same. For example:

```
http://mysatellite:1234/servlet/
ContentServer?pagename=myPage&favoritecolor=red
```

calls a different page than:

```
http://mysatellite:1234/servlet/
ContentServer?pagename=myPage&second_favoritecolor=blue
```

whether or not the content being generated is the same. Values passed in by the URL override values set in `pagecriteria`. For example, you have `myPage's pagecriteria` set to `red,blue`:

- If the URL passes in a value of `green,` then `green,blue` (not `red,blue`) will go into `myPage's` cache key.
- If the URL passes in values of `green,violet,` then `green,violet` (not `red,blue`) will go into `myPage's` cache key.
- If the URL passes in values of `green,violet,yellow`, an error results.

If a page does not have `pagecriteria` set, the values in the `resargs` fields go into the cache key. As with `pagecriteria`, values passed in by a URL override values specified in the `resargs` fields.

## Caching Properties

The default cache settings for Content Server and Satellite Server are contained in the `futuretense.ini` file. Additional Satellite Server properties are contained in `satellite.properties`. All properties can be modified by use of the Property Editor.

This section summarizes the Content Server and Satellite Server caching properties. For detailed information about the properties and the Property Editor, see the *Content Server Property Files Reference.*

### Content Server Page Caching Properties

The following properties in `futuretense.ini` control disk caching on Content Server:

- `cs.pgCacheTimeout`, which specifies the default timeout for pages in the Content Server cache.
- `cs.freezeCache`, which controls whether the cache pruning thread should run to remove expired entries from the cache.
- `cs.nocache`, which disables the entire Content Server page cache.
- `cc.SystemPageCacheTimeout`, which specifies the number of minutes a cached page is held in memory.
- `cs.alwaysUseDisk`, which specifies the default behavior for page entries in the SiteCatalog that have no cache override property specified.
- `cc.SystemPageCacheCSz`, which specifies the maximum number of pages that can be cached in memory.

### Satellite Server Properties

Satellite Server has two sets of properties (given in detail in the *Content Server Property Files Reference*):

- One set of properties is in the `futuretense.ini` file on your Content Server system and includes the following:
  - `satellite.page.cachecontrol.default`, a deprecated property that specifies a default value for the `cachecontrol` parameter for the `satellite.page`, and `RENDER.SATELLITEPAGE` tags and their JSP equivalents.
  - `satellite.blob.cachecontrol.default`, which specifies a default value for the `cachecontrol` parameter for the `satellite.blob`, and `RENDER.SATELLITEBLOB` tags and their JSP equivalents.
- The other set of properties is in the `satellite.properties` file on each Satellite Server host and comprises the following:
  - `cache_folder`, which specifies the directory into which Satellite Server will cache pagelets to disk.
  - `file_size`, which separates disk-cached pagelets and blobs from memory-cached pagelets and blobs according to the size that you specify.
  - `expiration`, which sets the default value for the length of time blobs stay in Satellite Server's cache.
  - `cache_check_interval`, which controls the frequency of the cache cleaner thread, and therefore when expired objects are pruned from cache.
  - `cache_max`, which specifies the maximum number of objects (pagelets and blobs) that can be cached (memory cache and disk cache combined) at a time.

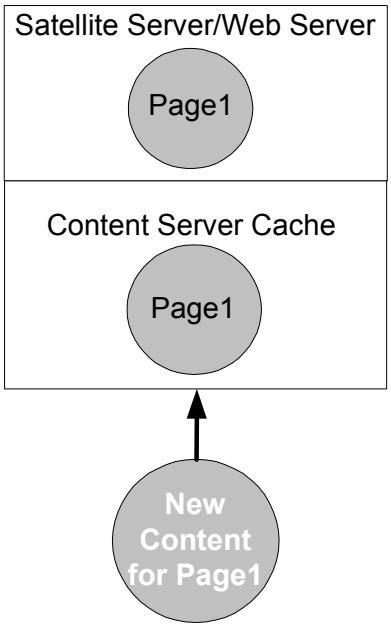
## Double-Buffered Caching

CS-Direct, CS-Direct Advantage, and Engage implement a double-buffered caching strategy, which uses the Content Server and Satellite Server caches in tandem on your live web site. This double-buffered caching strategy ensures that pages are always kept in cache, either on Content Server or Satellite Server.

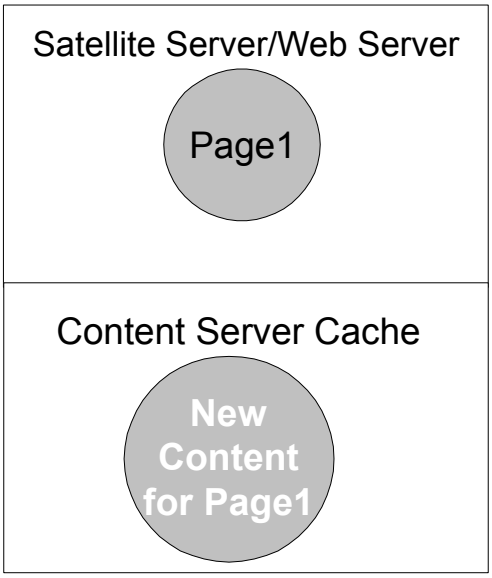
You can implement a similar caching strategy if you are running the Content Server core and Satellite Server without any of the other CS modules or products, by using the CacheManager Java API. For more information about the CacheManager Java API, see the *Content Server Javadoc.*

If you are running CS-Direct, both the Content Server core and Satellite Server caches are maintained by Content Server's CacheManager object. CacheManager tracks when content changes by logging elements and the assets that those elements call in cache tracking tables.

When assets are updated and published, the Content Server and Satellite Server caches are automatically flushed and updated in the following order:

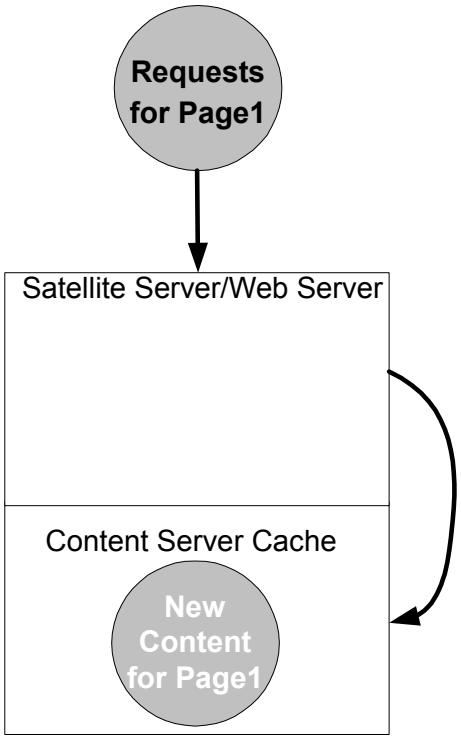



- Content providers publish updated assets to the delivery system. CacheManager checks the cache tracking tables to see which cached items are affected by the updated assets.

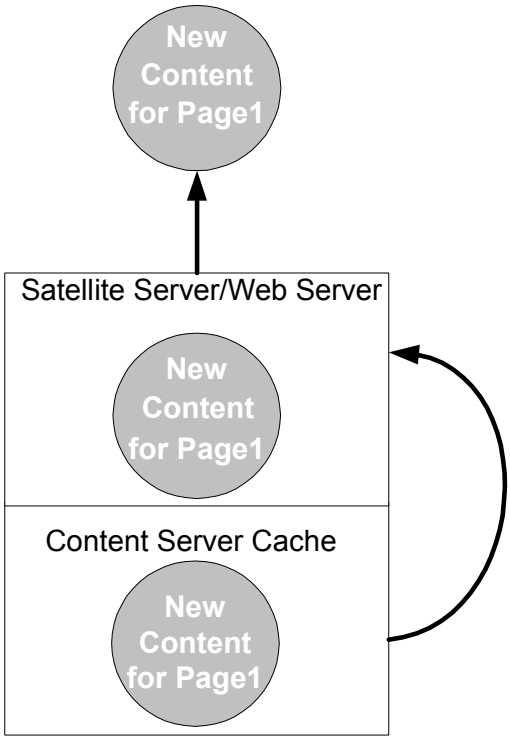



- CacheManager flushes the outdated `Page1` from the Content Server cache, then reloads the Content Server cache with the updated `Page1`.

  Any requests for `Page1` will be served the old version of `Page1` from the Satellite Server cache. This protects the Content Server machine from undue load as it deletes and rebuilds its cache.




- CacheManager flushes the outdated items from the Satellite Server cache. As visitors come to the web site and request `Page1`, the Satellite Server searches to see if `Page1` is in its cache. Because `Page1` is not in the Satellite Server cache, the request is passed on to Content Server.




- The Satellite Server system's cache is filled with an updated version of `Page1`, taken from the Content Server cache. The updated page is served to the requestors. If `Page1` were requested again, the page would be served from the Satellite Server cache.

## Implementing Double-Buffered Caching

The first step in implementing double-buffered caching on your web site is to design modular pages, as described in "Modular Page Design" on page 118. Once you have developed a modular page design, you implement a double-buffered caching strategy in three steps:

- Develop a pagelet caching strategy
- Set how individual pages and pagelets are cached by using the `pagecriteria` field of the `SiteCatalog` table
- Code your elements with Satellite tags

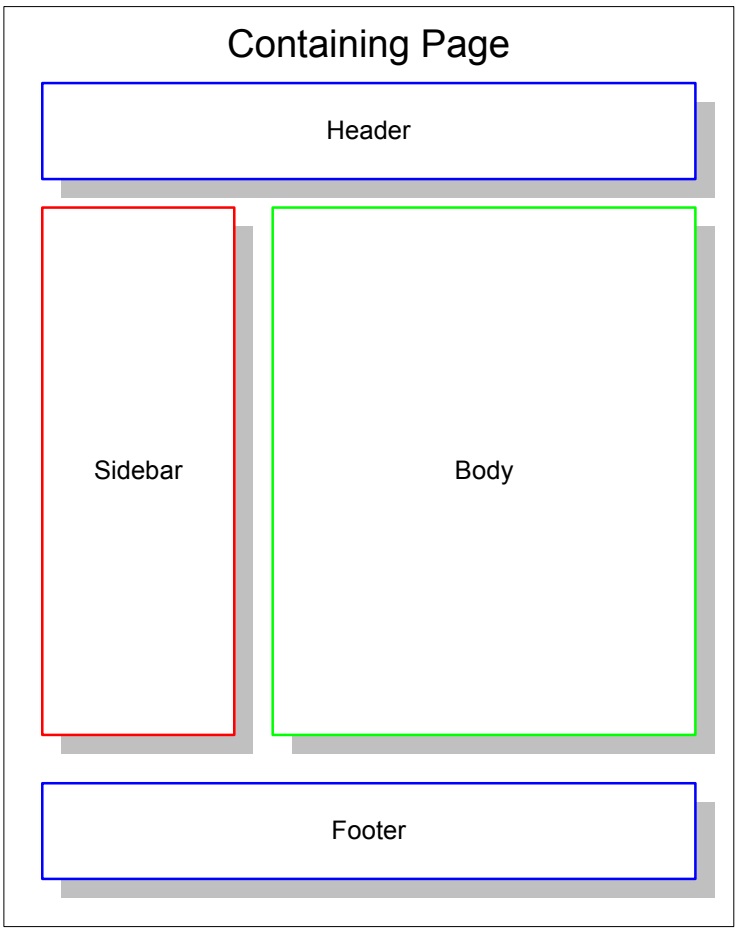
## Pagelet Caching Strategies

With a modular page design, caching occurs at the pagelet level; the containing page is never cached, so that any cached pagelets are always protected by ACLs. You choose which pagelets get cached based on how frequently they are updated.

The following table summarizes the guidelines for caching pagelets:

| Cache a Pagelet | Don't Cache a Pagelet |
|---|---|
| • If the content seldom changes.<br>• If the pagelet does not contain logic that requires evaluation to work. | • If the content changes frequently.<br>• If the content must be "real time."<br>• If the pagelet contains code that checks for ACLs, or other logic that requires evaluation to work. |

The following diagram is an example of a modular page:




The containing page should never be cached; this allows you to put logic, which requires evaluation by Content Server, into your pages, while still gaining the performance benefits of caching. It also allows your page to be protected by Content Server ACLs.

The header and footer pagelets in this example should be disk cached. They rarely gets updated, and should be designed accordingly. The header and footer may be static HTML written into your template, or disk-cached content from Content Server.

The sidebar is also a good candidate for disk caching. It has a small number of variations, and its content is determined by a small number of parameters.

Determining how to cache the body pagelet is more complex. The contents of the body pagelet probably depend on where the web site visitor is in the site. There are three possible types of content for the body pagelet:

- The results of a search that the web site visitor runs
- The results of a frequently run query
- An article

Your caching strategy should be as follows:

- If the content of the body pagelet is the result of a search based on parameters that the web site visitor enters, you do not want to cache it. Such pages change for each visitor, and there is little benefit to caching them.
- If the content is the product of a standard query that visitors often use, you should use resultset caching. Caching frequently run queries in the memory cache improves performance. For more information on resultset caching, see Chapter 14, "Resultset Caching and Queries."
- If the content of the body pagelet is the text of an article, you should cache the pagelet to disk.

## Setting cscacheinfo

The values in the cscacheinfo field of the `SiteCatalog` table allow you to control how pages get cached on Content Server on a page-by-page basis.

You can change these properties for each page and pagelet in your web site. For example, if you want a containing page element to be uncached on Content Server, set the values in `cscacheinfo` to `false`.

For more information on the cscacheinfo field, see "Creating Template Assets" on page 501.

## Coding for Caching

To implement double-buffered caching, you code your elements with Satellite Server tags. If you are running Content Server and Satellite Server only, use the Satellite tags documented in the Satellite Server sections of the *Content Server Tag Reference*.

Automatic cache maintenance is dependent upon logging your assets in the cache tracking tables. If you use the `ASSET.LOAD` tag to load an asset, that asset is automatically logged in the cache tracking tables. For those sections where `ASSET.LOAD` is not used, use the `RENDER.LOGDEP` tag to log content in the cache tracking tables.

**Note**

Cache dependencies are logged only if a page or pagelet is cached on Content Server. If a page is uncached on Content Server but cached on Satellite Server, that page will not be automatically flushed from the cache when its content is updated.

## Caching and Security

Cached pagelets require special security considerations as you design your site and develop your caching strategy. The following sections outline security considerations for pages cached in the Content Server and Satellite Server caches.

### Content Server Security

Pagelets that are disk cached on Content Server are bound by Content Server's ACLs, allowing you to use those ACLs to prevent unauthorized access to a page.

Note, however, that although Content Server checks the ACL of a containing page, it does not check the ACLs of the pagelets that the containing page calls. For example, suppose that your site uses three ACLs: `Open`, `Secret`, and `TopSecret`. Your containing page can be viewed by members of the `Open` ACL, but it calls pagelets that should be viewed only by members of the `Secret` and `TopSecret` ACLs. Because Content Server only checks a visitor's ACL of the containing page, visitors with the `Open` ACL can view content meant for members of the `Secret` and `TopSecret` ACLs.

#### To ensure that all the relevant ACLs are checked

1. Include the ACL for the page that you want to protect in that page's cache criteria, as shown in the following sample code:

```
<render.satellitepage pagename="innerwrapper"
userAcl="SessionVariables.member" c="Article" cid="123">
```

2. In the pagelet, insert code to check the ACLs, as shown in the following sample:

```
<asset.load name="art" type="Variables.c"
OBJECTID="Variables.cid"/>
<ASSET.GET NAME="art" FIELD="myACL"/> <!-- note you need a
column in your db to support this -->
<IF COND="Variables.userACL=Variables.myACL">
<THEN>
<render.satellitepage pagename="protected_art_tmpl1"
c="Variables.c" cid="Variables.cid"/>
</THEN>
<ELSE>
<render.satellitepage pagename="accessDenied"/>
</ELSE>
</IF>
```

## Satellite Server Security

Pagelets that will be cached on Satellite Server are bound only by Content Server ACLs under the following circumstances:

- If they are retrieved from the Content Server cache
- If they must be generated by Content Server to fulfill the page request

If a pagelet is served from the Satellite Server cache, it is no longer protected by Content Server ACLs.

To ensure that the content of your Satellite Server pages is secure, never cache your containing page and be sure that you put an ACL checking mechanism in the uncached container.

If your elements are coded with Satellite tags but you do not yet have Satellite Server installed, the page design considerations outlined in the Content Server Security section apply to you. Once Satellite Server is installed, however, Content Server checks the ACLs of uncached pagelets called from a containing page. The ACLs of pagelets cached on Satellite Server are not checked.

Chapter 6

# Intelligent Cache Management with Content Server

Whenever a site is built, it is critical that the rendering engine cache be properly configured so that all of the components work in concert. This chapter describes the rendering engine cache and its components. It also describes the cache configuration properties that enable CacheManager to clear all caches—ContentServer, BlobServer, and Satellite Server caches—of any object which becomes obsolete because of changes in its underlying content.

This chapter contains the following sections:


- Content Server's Rendering Engine Cache
- CacheManager
- Enabling CacheManager

# Content Server's Rendering Engine Cache

Content Server's rendering engine cache is a two-tier cache. Tier 1 consists of ContentServer and BlobServer; Tier 2 consists of Satellite Server. Each component is independently configurable with fine controls that tune cache size, cache timeout, and dependency management behavior.

Whenever a site is built, it is critical that the rendering engine cache be properly configured so that all of the components work in concert. If the components are configured correctly, Content Server can perform extremely well, effectively preventing users from viewing uncached content nearly all of the time. However, if these components are mis-configured, Content Server's behavior can be non-intuitive and unpredictable. Inadequate caching can hamper performance, and improper co-ordination of the cache inventory can result in stale content being rendered indefinitely. To address this, Content Server includes a module called CacheManager, which can actively manage the cache on behalf of the whole system.

# CacheManager

When a site uses CacheManager, it can record the existence of a "compositional dependency" against an object that is to be cached by the rendering engine. For example, if a pagelet renders an asset, then the asset is a compositional dependency on that page. If the asset changes, the page is no longer valid and must be flushed from cache.

Utilizing CacheManager to flush the cache requires ceding full control over the lifecycle of rendering engine cache objects to CacheManager by specifying that the objects never expire from the cache. When CacheManager determines that they are obsolete because of changes in the underlying content (i.e., in one of the compositional dependencies recorded against each object), it removes those objects from the cache.

**Note**

The reason for specifying an infinite expiration time is to ensure that CacheManager keeps a record of all objects that are cached, as well as what dependencies are tracked against them. This record is stored on Content Server, and it is linked to the existence of the cached object on the first tier. This record enables CacheManager to infer the existence of objects in the second tier cache and therefore flush the objects from the second tier cache.

If, however, an object were to expire from the cache, its record would be removed, leaving CacheManager without the information it requires to properly flush the object from the second tier cache.

Enabling CacheManager's features is almost completely automatic:

- By default, the cache is configured so that objects never expire.
- Compositional dependencies are recorded against the Blob and Page cache on the lower tier. Tags such as `<asset:load>` and `<render:sateliteblob>` provide *automatic* compositional dependency recording (see the *Tag Reference* for a complete list), whereas the two tags `<portal:logdep>`, and `<render:logdep>` provide *explicit* compositional dependency recording.

- Whenever assets are modified or published, CS-Direct automatically invokes CacheManager to purge the old content from the cache and, in the case of publishing, instructs CacheManager to pre-cache the new content in the background prior to flushing the second tier cache.

Site visitors enjoy the best possible cache performance by never having to view uncached content. For more information about recording compositional dependencies, consult the *Developer's Guide* and the *Tag Reference Guide*.

# Enabling CacheManager

This section describes the Tier 1 and Tier 2 cache configuration properties and how they must be set in order to enable CacheManager.

## Tier 1 Cache Configuration Properties

The tables in this section describe properties that regulate the Content Server page cache and BlobServer blob cache:

- Table 2, "Content Server Page Cache Properties"
- Table 3, "BlobServer Cache Properties"

More detailed descriptions are given in the *Property Files Reference*.

**Table 2:** Content Server Page Cache Properties

| Property | Description |
|---|---|
| `cs.pgCacheTimeout` | This property specifies how long a page should reside in the Content Server page cache, and it affects CacheManager as follows:<br><br>By default, this property is set to `0`, which means that the pages should reside in this cache forever, and that removal of these pages must be done explicitly. **For CacheManager to operate properly, this property must be set to `0`.** Otherwise, pages will expire, making it impossible for CacheManager to remove the corresponding pages from the Tier 2 cache, and users will view stale data.<br><br>Setting this property to a positive integer causes pages to expire after the number of minutes specified by the integer. |

**Table 2:** Content Server Page Cache Properties (continued)

| Property | Description |
|---|---|
| `cs.IItemList` | This property specifies the class implementing the `IItemList` interface, and it affects CacheManager as follows:<br>The `IItemList` interface is used to record compositional dependencies in the page cache. If this property is set to a legal class, then dependency items will be recorded against a page id in the `SystemItemCache` table, and this is what enables CacheManager.<br>An illegal value results in CacheManager having no effect. |
| `cc.SystemPageCacheCSz`<br>`cc.SystemPageCacheTimeout`<br>`cs.alwaysusedisk`<br>`cs.freezeCache`<br>`cs.nocache`<br>`cs.requiresessioncookies` | These properties are used to configure page caching, but have no effect on CacheManager. For more information about the properties, see the *Content Server Property Files Reference*. |

**Table 3:** BlobServer Cache Properties

| Property | Description |
|---|---|
| `bs.bCacheTimeout` | This property specifies how many seconds a blob should remain cached by BlobServer, and it affects CacheManager as follows:<br>When compositional dependencies are recorded against a blob in the `SystemItemCache` table, they are configured such that they will be removed from the table after the blob expires from the cache. This prevents excessive growth of the `SystemItemCache` table. However, removing the entry from the table disables CacheManager from removing the corresponding blobs from the Tier 2 cache, and users will view stale data. |
| `bs.bCacheSize` | This property specifies how many blobs will be stored in the BlobServer cache, and has no effect on CacheManager.<br>Unlike the `bs.bCacheTimeout` property, when a blob is evicted from the blob memory cache due to the cache being full, the corresponding row is not removed from the `SystemItemCache` table. Consequently, this property has no effect on CacheManager. |

**Table 3:** BlobServer Cache Properties (continued)

| Property | Description |
|---|---|
| `cs.recordBlobInventory` | This property specifies whether compositional dependencies should be recorded against blobs.<br>This property must be set to `true` (the default) for CacheManager to operate on blobs. |
| `bs.security` | This property controls the security feature of BlobServer, and it affects CacheManager as follows:<br>When BlobServer security is enabled, caching is disabled. Consequently, BlobServer security is incompatible with CacheManager's Intelligent Cache Management features.<br>By default, this level of security is disabled.<br>For more information about BlobServer security, see Chapter 5, "Page Design and Caching" and Chapter 7, "Advanced Page Caching Techniques." |
| `cs.manage.expired.blob.inventory` | This property controls the removal of blob dependency records from the `SystemItemCache` table after a blob expires from the blob cache. Its effect on CacheManager depends on the value of `bs.bCacheTimeout`.<br>• If `bs.bCacheTimeout` is set to 0 or less, this property has no effect.<br>• If `bs.bCacheTimeout` is set to a positive integer, setting this property to `true` ensures that CacheManager still operates correctly, but at the cost of growth in the `SystemItemCache` table. The default value is `false`. |

## Tier 2 Cache Configuration Properties

Tier 2 cache configuration properties deal with the Satellite Server cache, both page and blob.

None of the Tier 2 properties affect the correct operation of CacheManager. They do, however, serve as important diagnostic aids if CacheManager happens to be operating incorrectly. The timeout and configuration values of the Tier 2 cache properties are important in troubleshooting unpredictable behavior.

Typically, unpredictable behavior results when objects are cached on the Tier 2 cache but not on the Tier 1 cache, and so they are not actively flushed when the dependent asset is saved or published. (Consult the *Property Files Reference* for configuration details.)

Unpredictable behavior can also result if no compositional dependency is recorded against an object that is cached. This scenario precludes all active management of that object in the caches. Consult the *Tag Reference* for details about which tags automatically record

compositional dependencies, and which tags must be used in conjunction with explicit recording using one of the `:logdep` tags.

**Caution**

Do not record excessive compositional dependencies on your pages or blobs. Doing so will cause unnecessary flushing of the cache, which under certain circumstances, can result in severe performance problems during publishing. Be especially judicious when recording "unknown" compositional dependencies. Consult Chapter 27, "Coding Elements for Templates and CSElements" for more information about compositional dependencies.

Chapter 7

# Advanced Page Caching Techniques

Caching improves Content Server's performance in serving pages. Caching rendered content eliminates the need to render the content each time it is requested. This reduces the hardware requirements for the Content Server system, reduces the number of times clients make requests for uncached content, and ultimately improves response time.

The caching system has multiple layers, which allows regeneration of cached objects to be done on one cache level, while the client is being served cached content from another cache level. Content Server comprises the inner level of cache, and Satellite Server comprises the outer layer of cache.

This chapter describes how rendered object caching works in the Content Server platform. It describes how pages and blobs are cached, where they are cached, and how they are retrieved from cache on both Content Server and Satellite Server systems.

This chapter contains the following sections:


- Overview
- Configuring the Content Server Cache
- Configuring the Blob Server Cache
- Configuring the Satellite Server Cache
- CacheInfo String Syntax

# Overview

Both Content Server and Satellite Server cache pages, pagelets, and blobs. Content Server provides three different rendering engine caches: CS page cache, BlobServer cache, and SS cache. All the caches are controllable. They can be configured and emptied as follows:

- Maximum cache size can be configured. For information and instructions, see "Configuring Maximum Cache Size," on page 144 (Content Server cache), page 147 (BlobServer cache), and page 147 (Satellite Server cache).
- Objects can be stored in the cache with expiration information, such that when the object has expired from cache it is removed. For information and instructions, see "Setting Expiration Time for an Individual Entry," on page 145 (Content Server cache) and page 147 (BlobServer cache).
- Objects can be explicitly removed from the cache either manually, or automatically by using CacheManager. For information and instructions, see "Explicitly Removing Entries from Cache," on page 145 (Content Server cache), page 147 (BlobServer cache), and page 148 (Satellite Server cache).

# Configuring the Content Server Cache

There are two levels of caching for the Content Server page cache:

- In the database.
- In memory. Memory cache is a transparent subset of the database cache; however, it is independently configurable.

This section describes the main configuration settings for each cache.

## Configuring Maximum Cache Size

- Content Server's database cache does not have a maximum size configuration option.
- The memory subset of the page cache allows you to specify the maximum number of entries present by using the `cs.SystemPageCacheCSz` property in the `futuretense.ini` file. Setting a positive integer specifies the maximum number of entries that will be allowed to exist in the cache (the cache uses a Least Recently Used (LRU) algorithm for identifying the entry to be pruned when the maximum size has been reached).

**Note**

Setting a value of `0` will cause all entries to be added and then promptly removed. However, this should be avoided.

The `cs.SystemPageCacheCSz` property must not be set to `-1`.

The maximum size does not have anything to do with the aggregate number of bytes stored in cache.

## Setting Expiration Time for an Individual Entry

The lifetime of an individual entry in the CS page cache is determined by the `cscacheinfo` setting for each entry. The CacheInfo object derives values, if not explicitly set in the cscacheinfo field, from the configuration file. For CacheInfo syntax, see "CacheInfo String Syntax" on page 148.

## Explicitly Removing Entries from Cache

Content Server provides two ways of removing entries from cache: manually and automatically, using CacheManager.

### Manual Removal

You can manually remove an entry from the page cache, you can use the CacheServer servlet. The CacheServer provides two options:

- Flushing the entire cache.
- Forcing a flush of all pages at the moment they expire. In order to invoke CacheServer's "flush all" functionality, you must be logged in as a user with "destroy" privileges on the `SiteCatalog` table, and specify the parameter `all=true` when invoking the CacheServer servlet. If you do not specify a parameter, then all expired entries (those whose expiry date is in the past) will be cleared from the cache immediately. Entries that have not yet expired will not be cleared.

**Note**

In no case will an expired entry be served from the cache, even if it is still in the database table. Content Server checks the expiry date of any page it retrieves from cache before serving the page. If Content Server attempts to serve a page that has expired, the page will be removed from the cache immediately and a new page will be generated.

### Automatic Removal

CacheManager is a module that ties in very closely with the internals of the Content Server page and blob cache mechanisms. It is a tool that allows you to manage the contents of all of the rendering caches based on the items loaded on a page, on the expiration of the pages, or on parameters passed into pages.

CacheManager itself could be the subject of an entirely independent document. However, an overview of its functionality is described herein. For detailed information about its methods and required arguments, consult the COM.FutureTense.Cache.CacheManager JavaDoc.

1. A `CacheManager` is instantiated using one of two constructors. One constructor sets `CacheManager` with all of the currently registered Satellite Servers. The other constructor allows you to specify which Satellite Servers this instance of `CacheManager` will actually manage.
2. Next, the `CacheManager` needs to be populated with pages and blobs. This is done by using one of the following methods:

```
setByCachedDate(ICS ics, boolean before, String timestamp)
setByItemDate(ICS ics, boolean before, String timestamp)
setPagesByArg(ICS ics, String paramName, String paramValue)
```

`setPagesByID(ICS ics, String[] ids)`

The contents of the Page, Blob and Satellite caches are closely tied together. It is always the case, except as a result of a configuration error, that any object cached on Satellite Server will be present in the Content Server cache. This means that Content Server has a record of all entries in all rendering engine caches. **`CacheManager`** uses this record in order to be able to manage the contents of each of the caches, without having to directly interrogate each cache for the information explicitly.

`setByCachedDate(ICS ics, boolean before, String timestamp)`

This method allows you to populate CacheManager based on the date an entry was last added to the cache. You can choose whether you want to populate it with all of the entries modified either before or after the date specified.

`setByItemDate(ICS ics, boolean before, String timestamp)`

This method allows you to populate CacheManager based on the date an item on an entry was last modified. As with `setByCachedDate(ICS, boolean, String)`, you can choose whether you want all entries whose items were modified before or after the date specified.

`setPagesByArg(ICS ics, String paramName, String paramValue)`

This method allows you to populate CacheManager based on name-value pairs present in the cache key (including pagename).

`setPagesByID(ICS ics, String[] ids)`

This method allows you to populate CacheManager based on the exact item IDs of the items stored on the pages or blobs in the cache.

Once fully populated, CacheManager is able to manage the contents of the caches. This is done using one of the four main service methods:

- `flushCSEngine(ICS ics, int mode)`
  This method flushes all of the pages and blobs currently populated in the CacheManager from the Content Server page and blob caches.
- `flushSSEngines(ICS ics)`
  This method flushes all of the pages and blobs currently populated in the CacheManager from the Satellite Server cache. This is done by sending an http request to the FlushServer servlet with the appropriate `<page>` and `<blob>` tags embedded in it. Satellite Server interprets these tags and converts them into a cache key, then flushes the corresponding pages from cache.
- `refreshCSEngine(ICS ics, int mode)`
  This method sends a request (using `ICS.ReadPage` or `ICS.BlobServer`) that regenerates the object and automatically re-populates the cache
- `refreshSSEngines(ICS ics)`
  This method sends a request via http to Satellite Server to read the pages. The returned bytes are ignored, but the result is that the Satellite Server cache is re-populated.

Using these methods it is possible to take advantage of double-buffered caching, a tool that can enable extremely high performance dynamic sites. For more information about double-buffered caching, see “Double-Buffered Caching” on page 128.

# Configuring the Blob Server Cache

The BlobServer cache is an “all or nothing” cache—entries are either globally cached or globally not cached.

BlobServer caching is disabled if security is enabled. Thus, if `bs.security=true`, caching is disabled.

## Configuring Maximum Cache Size

The property `bs.bCacheSize` in `futuretense.ini` specifies the number of entries the blob cache will contain. If the size is set to a negative number, the blob cache will be allowed to grow indefinitely.

## Setting Expiration Time for an Individual Entry

Blob Server does not support individual entry expiration for cached entries. All cached objects will reside in cache for the timeout determined by the `bs.bCacheTimeout` property in `futuretense.ini`. A negative timeout indicates that entries should not time out. A positive integer specifies the number of minutes an object will reside in cache.

## Explicitly Removing Entries from Cache

BlobServer supports the flushing of both individual entries and all entries from the cache.

### Manual Removal

To manually remove an entry from cache, simply rename the `blobtable` parameter to `flushblobtable`. This will remove the entry corresponding to the rest of the parameters from the cache.

To manually remove all entries from the cache, there are two options. One is to invoke the BlobServer servlet with the parameter “`flushblobtables`” (notice the “`s`”). The other is to invoke the CacheServer servlet as described above. However, this will flush all pages and all blobs from the cache.

### Automatic Removal

Because blob dependency items are recorded when blob links are generated, it is possible to invoke CacheManager to manage blobs as well as pages. (In fact, CacheManager always manages blobs and pages together). Refer to the sections about CacheManager and Content Server for details about using CacheManager.

# Configuring the Satellite Server Cache

The generic Satellite Server cache configuration is done in the `satellite.properties` file; typically, however, cache configuration is overridden on an object-by-object basis.

## Configuring Maximum Cache Size

The maximum number of entries that can be stored in the cache at once is configurable using the `cache_max` property in the `satellite.properties` file. If the property is set to a negative integer, the cache will not be limited by size. Any positive integer will specify the maximum number of entries that can be stored in the cache.

## Explicitly Removing Entries from Cache

Individual entries can be removed from the Satellite Server cache either manually or using CacheManager, as explained in this section.

### Manual Removal

Satellite Server includes a servlet called FlushServer. By submitting a `GET` request to this servlet (specifying the username, password and reset parameters), it is possible to flush all of the contents of the Satellite Server cache. It is not possible to flush individual entries using `GET`.

### Automatic Removal

As described above, it is possible to flush the Satellite Server cache using CacheManager. As described, CacheManager is only able to flush entries on Satellite Server if a corresponding object is cached on Content Server. This is the case because of the way Content Server tracks the contents of the Satellite Server cache.

As described above, it is possible to flush the Satellite Server cache by using CacheManager, as long as a corresponding object is cached on Content Server. The corresponding object is required because of the way Content Server tracks the contents of the Satellite Server cache.

The relevant CacheManager methods for dealing with the Satellite Server cache are `flushSSEngines()` and `refreshSSEngines()`. For information about the methods, see page 146.

# CacheInfo String Syntax

The `cscacheinfo` and `sscacheinfo` fields of the SiteCatalog are populated with a CacheInfo string. This section describes the format of the string. It is a two-part, comma-separated string. The first part indicates whether the page will be cached. The second part describes the expiration.

Sample values:

```
false
true
true,*
true,~4
true,@1987-06-05 04:32:10
true,#00:00:00 */*/*
*
(blank)
```

## CacheInfo String: First Part

The first part in CacheInfo must be one of the following values:

```
false
true
(blank)
*
```

- If the value is `false`, then the page will not be cached.

- If the value is `true`, then the page will be cached according to the information provided in the second element.
- If the value is blank, then Content Server will consult the `futuretense.ini` property `cs.alwaysusedisk`. If this property is set to `yes`, then a blank value will be interpreted as having the same behavior as `true`. If the value is set to `no` (the default value), then a blank value will be interpreted as having the same behavior as `false`.
- If the value is *, then it will be treated as blank.

## CacheInfo String: Second Part

The second part in CacheInfo describes when a page that is to be cached should be removed from cache. If the first element is `false` (or is interpreted as `false`), then the second element is ignored.

There are three ways of specifying the expiration of a page:

```
page timeout (in minutes)
instant in time expiration
cron-like TimePattern expiration
```

Legal values include:

```
~<number of minutes>
@<date in JDBC format>
#<COM.FutureTense.Util.TimePattern format>
*
(blank)
```

## Page Timeout

If the second element starts with ~, then the value following the ~ must be an integer. The value of this integer is the number of minutes a page will remain in cache after it was first created. A negative value or “0” indicates that the page will never expire (it will remain in cache forever).

## Absolute Moment in Time

If the second element starts with `@`, then the value following the `@` must be a date expressed in the JDBC date string format, namely, `YYYY-MM-DD HH:MM:SS`. Once that date has passed, cached pages will be flushed from cache and the page will no longer be cached.

## TimePattern

The TimePattern format is supported for describing page cache expiration. If the second element starts with `#`, then the value following the `#` must be a valid TimePattern string as defined by the public class `COM.FutureTense.Util.TimePattern`.

In general, the TimePattern syntax corresponds to the format used in most UNIX cron tables. It allows you to specify expiration at a specific time or times every day, month, week, day of week, and year.

It is expected that the TimePattern format will become the most widely used format for page expiration.

## Wildcard

If the second element is *, then the page will assume a timeout expiration behavior, as described in Timeout above. The timeout value will be read from the `futuretense.ini` file's `cs.pgCacheTimeout` property.

## Blank

If the second element is blank, then it assumes the same behavior of *.

Chapter 8

# Content Server Tools and Utilities

Content Server includes several tools and utilities that you use together with the Content Server browser-based interface for developing and maintaining your web sites. This chapter provides brief descriptions of these utilities, and tells you how to start them. It includes the following sections:

- Content Server Explorer
- CatalogMover
- Property Editor
- XMLPost

# Content Server Explorer

The Content Server Explorer tool is a Microsoft Windows application for viewing and editing tables and rows in the Content Server database, and for creating and editing executable elements (or files) written in XML or JSP. You use Content Server Explorer to do the following:

- Add entries to tables
- Edit rows within tables
- Track revisions to rows of tables
- Create and drop Content Server tables
- Organize tables and folders into projects
- Preview `SiteCatalog` records as pages in a browser
- Export and import records as integrated `.cse` type files
- Export and import tables and projects in `.zip` files

Content Server Explorer is installed along with Content Server.

## Connecting to a Content Server Database

You can use Content Server Explorer on any remote Microsoft Windows machine simply by copying the Content Server Explorer directory on a machine where Content Server is installed (`tools/ContentServerExplorer`) to a directory on the remote machine. You then start the Content Server Explorer executable file (`ContentServerExplorer.exe`) and log in to Content Server by supplying a user name, password, hostname, port, and protocol information.

### To connect to a system that is running Content Server

1. Start Content Server Explorer.
2. Choose **File > Open Content Server** to display the Login dialog box.
3. Enter the following values:

   **Name** – Your Content Server user name.

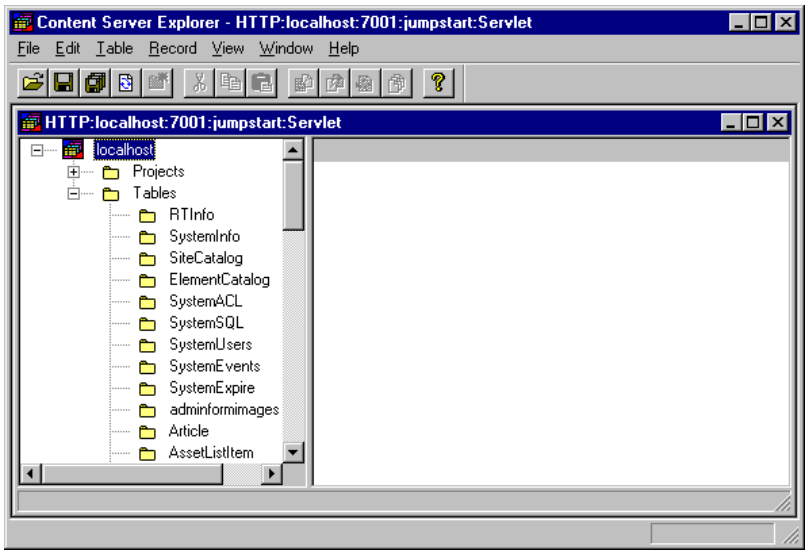
   **Password** – Your Content Server password. (Depending on your site security, it may not be necessary to enter a name and password.)

   **Host name** – The hostname or IP address. You cannot leave this field blank.

   **Port** – The port number (the default is 80).

   **Protocol** – Typically, this is HTTP. (You may select HTTPS if the web server is running SSL.)

   **Application server URL path** – The type of application server for your site.
4. Click **OK** to log in. The Content Server Explorer window appears:




You may want to create a shortcut on your Windows desktop to Content Server Explorer. For instructions about using Content Server Explorer, see the online help as well as sections in this manual that describe specific tasks requiring Content Server Explorer. For more information on Content Server Explorer and its features, see the Content Server Explorer online help.

# CatalogMover

You use the CatalogMover tool to export and import Content Server database tables, including the `ElementCatalog` and `SiteCatalog` tables. For example, you can use CatalogMover to export page elements and content assets to one system, and load the same elements and assets into the database on another system. You can export and import database tables as either HTML files or ZIP files.

You can use CatalogMover through either the Windows interface described in the following sections, or the command line interface described in "Command Line Interface" on page 160.

**Note**

In previous versions of Content Server, tables in the Content Server database were called "catalogs." This term still applies to the names of some database tables as well as to the CatalogMover tool itself.

## Starting CatalogMover

### To start CatalogMover

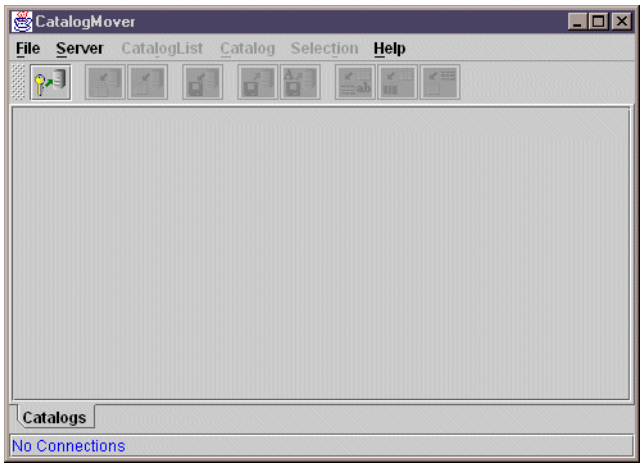
Execute the following scripts at the MS DOS prompt or in a UNIX shell:

- Windows: `CatalogMover.bat`
- Solaris: `CatalogMover.sh`

The following JAR files must be in the classpath, or be specified by the `-classpath` switch:

```
cs.jar
swingall.jar
commons-logging.jar
cs-core.jar
```

The CatalogMover window appears:




## Connecting to Content Server

Before using CatalogMover, you must first connect to a Content Server system.

### To connect to Content Server

1. Choose **Server** > **Connect**. The Connect to Server dialog box appears.
2. In the **Server** field, enter the name of the HTTP server you want to connect to, and the port on which the server is running.
3. In the **Name** field, enter your user name.
4. In the **Password** field, enter your password.
5. Select one of the following radio buttons:
   - **Standard Servlets** – to connect to a system using WebSphere or WebLogic.
   - **Sun ONE Application Server** – to connect to a system using Sun ONE Application Server.
   - **Custom** – to connect to a different application server, enter the following value in the text box:

     `<ft.approot><ft.cgipath>/CatalogManager.`
6. Click **Connect**.

## CatalogMover Menu Commands

CatalogMover includes the following menu commands:

## File Menu

- **Exit** – Disconnect from Content Server and close CatalogMover.

## Server Menu

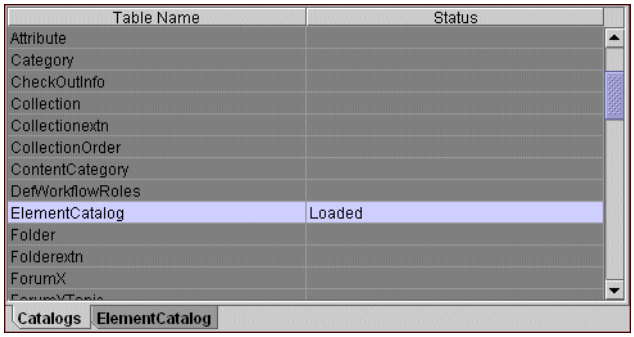
- **Connect** – Display the Connect to Server dialog box.
- **Reconnect** – Display the Connect to Server dialog box and renew the current Content Server connection.
- **Disconnect** – Disconnect from Content Server.
- **Purge Temporary Tables** – Purge imported tables before committing.
- **Commit Individual Tables** – Commit imported tables to the database.
- **Normalize Filenames on Export** – Enable CatalogMover's file name normalization behavior, which changes the names of files that are being moved to names that match their corresponding ID numbers. If this feature is not enabled, file names are not altered.

## CatalogList Menu

- **Load** – Display a list of all tables in the database.

## Catalog Menu

- **Load** – Load into local memory a table from the list. The following figure shows a loaded `ElementCatalog` table:




  Click the **Element Catalog** tab to view all rows in the table, and to select specific rows for export.

- **Refresh** – Update the loaded tables from the Content Server database.
- **Auto Import Catalog(s)** – Import a previously exported ZIP file.
- **Import Catalog** – Import into the local database a table that was exported from another Content Server database.
- **Export Catalog Rows** – Export the selected rows in the loaded table.

## Selection Menu

- **Select All Rows** – Select all rows in the currently displayed table.
- **Deselect All Rows** – Deselect all rows in the currently displayed table.

- **Select Rows By SubString** – Select rows in the currently displayed table by typing a portion of any field value string that uniquely identifies a set of rows.

### Help Menu

- **About** – Display version information about the Content Server installation.

## Exporting Tables

**Exporting** is the process of retrieving table rows and their content from the database and saving them in local HTML files and associated data directories. CatalogMover creates one HTML file per table.

#### To export selected table rows

1. Connect to Content Server as described in "Connecting to Content Server" on page 154.
2. Choose **CatalogList > Load** to display a list of all tables in the database
3. Choose **Catalog > Load** to load a table, and select rows as described in "Selecting Rows for Export" below.
4. Choose **Catalog > Export Catalog Rows**.

   A dialog box appears prompting you to specify a directory for the HTML file containing the exported rows.
5. Navigate to your directory of choice, and click **Save**.

   CatalogMover exports the selected rows to your selected directory.

## Selecting Rows for Export

You can select specific rows for export in a loaded table by clicking on them, or you can search for specific rows by substring.

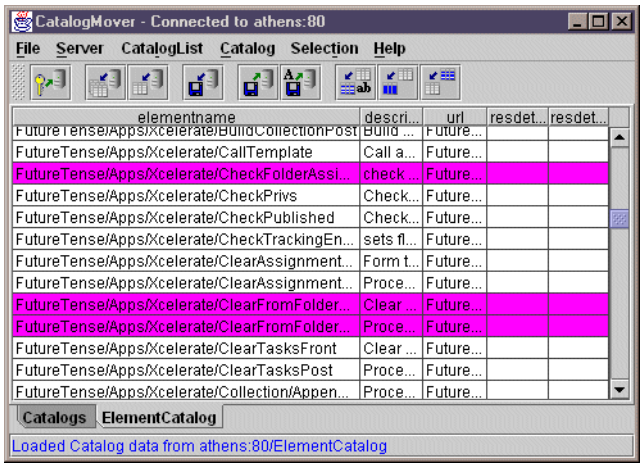
#### To search for and select rows according to a substring

1. Choose **Selection > Select Rows By SubString**.

   The following dialog box appears:

2. In the text field, enter the substring you want to locate. For example, if you wanted to search the ElementCatalog primary key for all rows with "folder" in the element name, enter `folder` and click **OK**.

   CatalogMover searches the table and selects the rows that match your substring query against the primary key for the table, as shown in the following figure:




**Note**

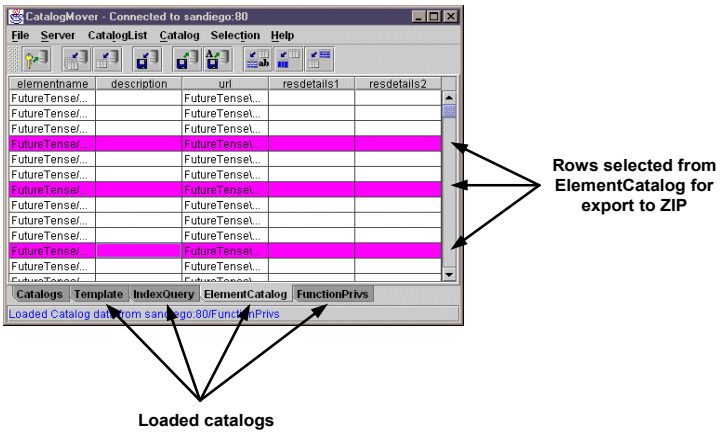
Selecting rows by substring only works for the left-most column in the table. However, you can change column positions so that any column can become the left-most column. To do this, simply click and drag the column header.

## Exporting to a ZIP File

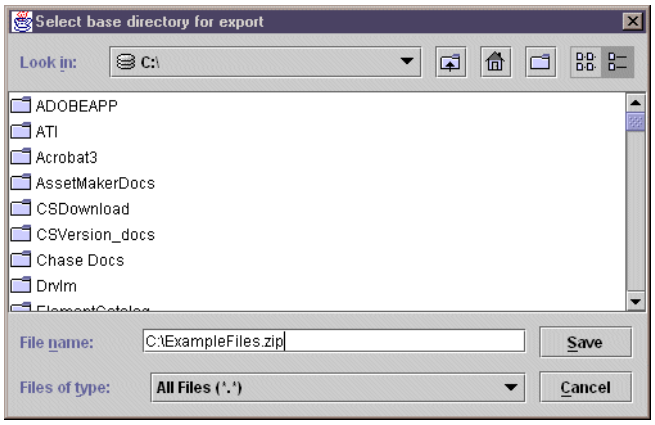
You can select several rows from several tables and export them to a ZIP file on local machine from which you are running CatalogMover. Once you create the ZIP file, you can import the contents of the file into server tables.

### To export a ZIP file with CatalogMover

1. Choose **CatalogList > Load** to display a list of all tables in the database.
2. Choose **Catalog > Load** to load a table, and select rows as described in "Selecting Rows for Export."




3. Choose **Catalog > Export Catalog Rows**. The following dialog box appears:



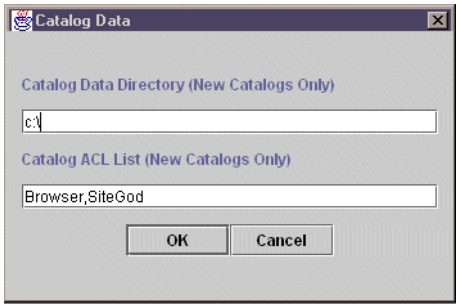

4. Navigate to the directory where you want to save the ZIP file.
5. In the **File Name** field, enter a name for the files and type a .ZIP file extension.
6. Click **Save**. The rows you selected from all of the tables are exported to a ZIP file in the directory you chose.

## Importing Tables

**Importing** is the process of sending locally stored HTML files and the associated data to the server. You can select a particular HTML file to import, or you can choose to import all HTML files.

### To import HTML files that have been previously exported from another table

1. Connect to the Content Server installation you want to import the HTML files to, as described in "Connecting to Content Server" on page 154.
2. Choose **CatalogList > Load** to display a list of all tables in the database.
3. Choose **Catalog > Import Catalog**.
4. Navigate to the HTML file containing the previously exported table rows.
5. Select the HTML file and click **Open**. The following dialog box appears:




6. If you are importing new table rows that do not currently exist, enter the information in the **Catalog Data Directory** and the **Catalog ACL List** fields.

   If you are replacing existing table rows with the imported table rows, leave these fields blank.

7. Click **OK**. The table rows contained in the previously export HTML file are imported into the Content Server database to which you are connected.

   A dialog box appears, listing the table rows that were imported.

> **Note**
>
> If you import tables that do not exist on the server to which you are connected, the new tables are automatically created as they are imported.

## Importing a Previously Exported ZIP File

You can import table rows stored in an exported ZIP file to your server using CatalogMover.

### To import a previously exported ZIP file

1. While connected to your database, choose **Catalog > Auto Import Catalogs**.
2. In the resulting dialog box, navigate to the directory where you previously exported the table rows. To see the ZIP file, change the **Files by Type** drop-down menu to **all files**.
3. Select the ZIP file and click **Save**. The rows contained in the ZIP file are automatically imported to your database.

## Merging Existing CatalogMover Files

### To merge CatalogMover files

1. Connect to the Content Server installation you want to import the HTML files to, as described in "Connecting to Content Server" on page 154.
2. Choose **CatalogList > Load** to display a list of all tables in the database.
3. Choose **Catalog > Load** to load a table, and select the rows that you want to merge into another file, as described in "Selecting Rows for Export".
4. Choose **Catalog > Export Catalog Rows.**
5. Navigate to the HTML file you want to merge the rows with. Click **Save**. The following dialog box appears:

6. Click **Update existing exported data**. CatalogMover merges the exported rows into the HTML file you selected.

## Replacing Existing CatalogMover Files

### To replace CatalogMover files

1. Connect to the Content Server installation you want to import the HTML files to, as described in "Connecting to Content Server" on page 154.
2. Choose **CatalogList > Load** to display a list of all tables in the database.
3. Choose **Catalog > Load** to load a table, and select the rows that you want to merge into another file, as described in "Selecting Rows for Export."
4. Choose **Catalog > Export Catalog Rows**.
5. Navigate to the HTML file you want to merge the rows with. Click **Save**. The following dialog box appears:

6. Click **Replace existing exported data**. CatalogMover replaces rows in the HTML file you selected with the exported rows.

## Command Line Interface

The following parameters allow CatalogMover to perform functions without displaying a GUI. The parameter is followed by a space followed by the value:

| Parameters | Description |
|---|---|
| `-h` | display command line parameters |
| `-u` *`username`* | username |
| `-p` *`password`* | password |
| `-s` *`servername`* | servername to connect |
| `-b` *`baseurl`* | base URL – either `http://($host)/cgi-bin/gx.cgi/AppLogic+FTCatalogManager (NAS)` or `http://($host)/servlet/CatalogManager(WebLogic)` |
| `-t` *`table`* | table name – used when exporting to designate tables to export, use multiple -t parameters to export multiple tables |
| `-x` *`function`* | function to perform – legal values are `import`, `import_all`, `export`, `export_all` |
| `-d` *`directory`* | directory – When exporting, directory to contain exported tables. When importing all, directory containing all tables to import. |
| `-f` *`filename`* | file containing table to import – Can either be an HTML file or a ZIP file generated by export. |
| `-c` *`directory`* | upload directory to be used if creating a table |

| Parameters | Description |
|---|---|
| `-a aclone,acltwo,...` | ACL list – comma-separated list of ACLs to be used if creating a table |

# Property Editor

The Property Editor tool provides an easy-to-use Windows interface that lets you view, modify, and add properties in the Content Server `futuretense.ini` file.

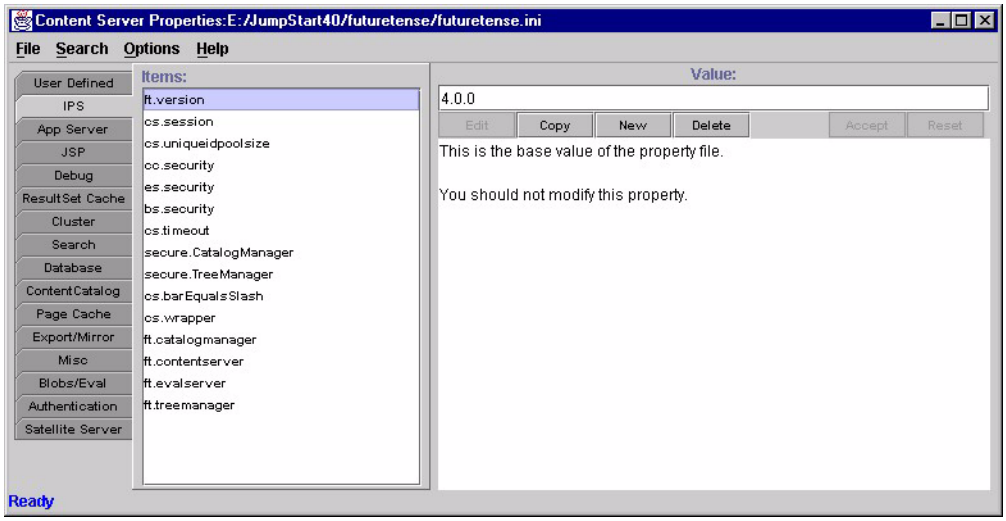
## Starting the Property Editor

### To start the Property Editor

Run the following scripts:

- On Windows NT: `propeditor.bat`
- On Solaris: `propeditor.sh`

The Properties window appears:




The Properties window displays properties in functional groups, such as Database and Caching, on the left side of the window.

The **Items** pane lists the properties in the selected functional group.

The **Value** pane lists the current value for the selected item, a brief description of the item, and the acceptable values for it.

**Note**

The `futuretense.ini` file contains a release number string, `ft.version`, which contains a value such as 4.0.0. that is set by Content Server.

Do not modify this property—it is for reference only.

## Setting Properties

**To set Content Server properties on your system**

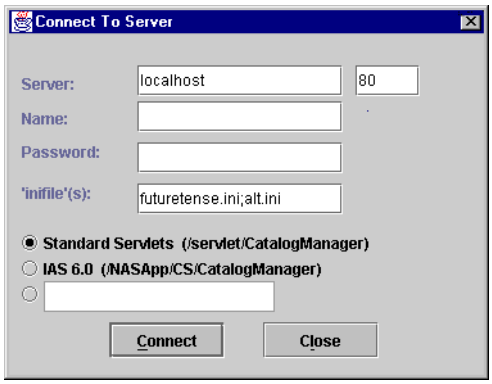
1. If necessary, start the Property Editor.
2. Choose **File > Search** and open the `.ini` file you want to edit.
3. Select a properties group from the tabs on the left side of the window. The Property Editor displays the properties in the **Items:** pane.
4. Select a property in the **Items:** pane. The Property Editor displays the current property value and a brief description in the **Values:** pane.
5. In the **Values:** pane, enter the new value in the text field.
6. In the **Values:** pane, click the **Accept** button.
7. Repeat steps 3 through 6 for all the properties you want to change.
8. When you finish, choose **File > Save** to save your changes.
9. Click **OK** in the confirmation message box.
10. Choose **File > Save** to save your changes and close the Property Editor.
11. Stop and restart the application server to apply the changes.

## Merging Property Files

You ordinarily use the Property Editor to modify the `futuretense.ini` property file. You can also add properties to your property file from the property file on the server to which you are connecting. If you have the same properties with different values defined in multiple .ini files, Content Server uses the values in the last property file that it loads.

To merge a property file with the `futuretense.ini` file, enter the names of the two property files separated by a semicolon in the **'inifile'(s)** field when you connect to Content Server. If you do not want to merge property files, leave this field blank.

For example, the following Connect to Server dialog box shows a merge between `futuretense.ini` and `alt.ini`:



# XMLPost

The XMLPost utility imports data into the Content Server database. This utility is based on the Content Server FormPoster Java class and it is delivered with the Content Server base product. It imports data using the HTTP POST protocol.

To import assets, you use XMLPost with posting elements that are delivered with Content Server. For information about using XMLPost to import assets, see the following chapters:

- Chapter 20, “Importing Assets of Any Type”
- Chapter 21, “Importing Flex Assets”

Chapter 9

# Sessions and Cookies

This chapter explains how to use XML tags to manage sessions and cookies. It contains the following sections:


- What Is a Session?
- Session Lifetime
- Sessions Example
- What Is a Cookie?
- Cookie Example
- Tips and Tricks
- Satellite Server Session Tracking

# What Is a Session?

Imagine a web site containing two pages: `main` and `water`. Suppose a visitor sees `main` first and then moves on to `water`. HTTP is a stateless protocol. So, if a typical web server is managing this site, any knowledge gathered at `main` is lost when the visitor browses over to `water`. In other words, `water` cannot take advantage of any information that the visitor might have provided at `main`.

To get around this limitation, application servers detect when a visitor first enters a web site. At that point, the application server starts a **session** for this visitor. In the preceding example, when the visitor requests the `main` page, the application server starts a session. The web site designer can use `main` to gather information about the visitor and store that information in **session variables**. The information in session variables is available to all subsequent pages. So, for example, if Bob provides his age to `main`, and `main`'s designer wrote the age to a session variable, then `water` could easily access Bob's age.

**Session variables** contain values available for the duration of the session. When the session ends, the application server destroys the session variables associated with that session. Each session variable consumes memory on the application server, so creating unnecessary session variables can hurt performance.

Content Server automatically creates some session variables; the web site developer can optionally create others.

The application server can maintain sessions on a cluster.

# Session Lifetime

A session begins when a visitor first hits your web site. The session ends when any of the following happens:

- The visitor terminates his browser.
- The session has timed out. The `cs.timeout` property is used by Content Server to set the session timeout value in the application server. If this property is set to 300, then a user session becomes invalid in 300 seconds, or 5 minutes.
- The system administrator stops the application server.

## Session Variables Maintained by Content Server

Upon creating a session, Content Server automatically creates the following session variables:

| Session Variable | What it Holds |
|---|---|
| `SessionVariables.currentUser` | The `id` of the visitor logged in. |
| `SessionVariables.currentAcl` | The comma-separated list of all ACLs to which this visitor belongs. If the visitor has not explicitly logged in, the default ACL is `Browser`. |
| `SessionVariables.username` | The username under which this visitor is logged in. If the visitor has not explicitly logged in, the default username is `DefaultReader`. |

| Session Variable | What it Holds |
|---|---|
| `SessionVariables.iniFile` | The name of the file containing Content Server properties. |

## Logging In and Logging Out

When a visitor first hits the site, Content Server creates a session and implicitly logs in the visitor as `DefaultReader`. During the session, if the visitor explicitly logs in, Content Server automatically updates the values of `SessionVariables.currentUser`, `SessionVariables.currentAcl`, and `SessionVariables.username`. Logging in does not affect the values of any other session variables. In other words, if your pages create session variables prior to a login, then those values are still valid after the login. When a visitor explicitly logs out, the Content Server-generated session variables automatically revert to the values they held prior to login. For example, consider the following sequence:

1. A visitor first hits a page, so the value of `SessionVariables.username` is `DefaultReader`.
2. The visitor logs in as marilyn, so the value of `SessionVariables.username` is `marilyn`.
3. If marilyn logs out, the value of `SessionVariables.username` reverts to `DefaultReader`.

To trigger a logout, you call the `<CATALOGMANAGER>` tag with the `ftcmd=logout` modifier. When issuing this tag, you can optionally supply the `killsession` modifier, which destroys the current session. You can then create a new session by invoking the `<CATALOGMANAGER>` tag with the `ftcmd=login` modifier.

# Sessions Example

Here's a simple session example, consisting of three very short elements:

| Element | What it Does |
|---|---|
| `FeelingsForm` | Asks visitors to pick their current mood. |
| `SetFeelings` | Assigns the current mood to a session variable. |
| `Meat` | Evaluates the session variable. |

## FeelingsForm Element

The feelings form doesn't really involve sessions or variables; this element merely generates a form. The visitor's chosen mood is passed to the `SetFeeling` element:

```
<form action="ContentServer" method="post">
   <input type="hidden" name="pagename"
          value="CSGuide/Sessions/SetFeelings"/>
```

```
 <P>How are you feeling right now?</P>
 <P>
 <select name="Feeling" size="1">
    <option>Good</option>
    <option>Not so Good</option>
 </select>
 </P>

 <P><input type="submit" name="doit" value="Submit"/></P>
</form>
```

The resulting page looks like the following:

How are you feeling right now?

Not so Good

Submit

## SetFeeling Element

Upon clicking the Submit button, the visitor is transported to `SetFeeling`. This element assigns the visitor's mood to a new session variable named `CurrentFeeling`.

```
<SETSSVAR NAME="CurrentFeeling" VALUE="Variables.Feeling"/>

<P>Welcome to our site.</P>

<P>Now proceed to
<A href="ContentServer?pagename=CSGuide/Sessions/Meat">
some meaty content.
</A></P>
```

The resulting page looks as follows:

```
Welcome to our site.

Now proceed to some meaty content.
```

If an element in this application asked the visitor to login, Content Server would have automatically set the `username` session variable to the visitor's login name. In that case, you could have personalized the welcome message in `SetFeeling` as follows:

```
<P>Welcome to our site, <CSVAR
NAME="SessionVariables.username"/>
</P>
```

## Meat Element

Upon clicking some meaty content, the visitor is transported to the `Meat` page. This page evaluates the session variable:

```
<IF COND="SessionVariables.CurrentFeeling=Good">
 <THEN>
     <P>Sessions are happiness.</P>
```

```
  </THEN>
  <ELSE>
      <P>Don't let sessions get you down.</P>
  </ELSE>
 </IF>
```

A visitor in a not so good mood sees:

```
Don't let sessions get you down.
```

Notice how `CurrentFeeling` was available to `Meat`. In fact, `CurrentFeeling` is available to any other elements in the session.

# What Is a Cookie?

A **cookie** is a string that your application writes to the visitor's browser. A cookie stores information about visitors that lasts between sessions. The visitor's browser writes this string to a special cookie file on the visitor's disk. When that visitor returns to your web site, the visitor's browser sends a copy of the cookie back to the web server that set it. Once a cookie has been created, it is available as a variable to elements on a page.

For example, your application might store the visitor's favorite sports team in a cookie. Then, when the visitor returns, your application could retrieve the cookie and use its information to display the team logo in a banner.

When cookies are no longer needed, you can delete them.

## CookieServer

CookieServer is a servlet that sets cookies for you. You access CookieServer by creating cookies with the `satellite.cookie` tag.

## Cookie Tags

Content Server offers two tags for managing cookies:

| Tag | Use |
|---|---|
| `satellite.cookie` | Sets a cookie on the client's browser. |
| `REMOVECOOKIE` | Deletes a cookie from the client's browser. |

There is no special tag to obtain the value of a cookie. Instead, when a visitor returns to the web site, Content Server loads the value of the cookie as a regular variable.

When creating a cookie (by calling `satellite.cookie`), you can specify the following attributes:

| Attribute | Value |
|---|---|
| `name` | Name of the cookie. This also serves as the name of the incoming variable containing the value of the cookie.<br>**Warning:** Cookies in the CS page context are treated as variables. Therefore, when a cookie and an asset attribute share the same name, they are treated as the same variable. |
| `expiration` | Time in seconds after which the cookie no longer is sent to the web server. |
| `security` | Optionally set security on the cookie. |
| `URL` | Restrict that the cookie only be sent on this URL |
| `Domain` | Restrict that the cookie only be sent to URLs in the specified domain. |

Because they feel that cookies are a security threat, some visitors configure their browsers to reject cookies. If the information in the cookie is critical, your application must be prepared for this.

You must set or remove cookies before using any tags that stream content back to the visitor's browser. You must set or remove cookies even before the `<HTML>` tag.

# Cookie Example

This example consists of several very short elements:

| Element | What it Does |
|---|---|
| `Start` | Determines whether a cookie is set. If cookie is set, call `DisplayWelcome`. If cookie is not set, call `GetColorPreference`. |
| `ColorForm` | Displays a form that asks visitor to pick her favorite color. |
| `CreateCookie` | Creates a cookie on this visitor's browser. Then, redirects visitor to `DisplayWelcome`. |
| `DisplayWelcome` | Displays a simple welcome message in the visitor's favorite color. |

## Start.xml

The `Start.xml` element determines whether the cookie has already been set. If the cookie has been set, Content Server stores its value inside a regular variable named `Variables.ColorCookie`. The code for `Start.xml` is as follows:

```
<IF COND="IsVariable.ColorCookie=true">
```

```
 <THEN>
    <CALLELEMENT NAME="CSGuide/Sessions/DisplayWelcome"/>
 </THEN>
 <ELSE>
    <CALLELEMENT NAME="CSGuide/Sessions/ColorForm"/>
 </ELSE>
</IF>
```

## ColorForm

The `ColorForm.xml` element displays some an HTML form to gather the visitor's favorite color. The code for `ColorForm.xml` is as follows:

```
<form action="ContentServer"
     method="post">
   <input type="hidden" name="pagename"
          value="CSGuide/Sessions/CreateCookie"/>

 <P>What is your favorite color?</P>
 <P>
 <select name="FavoriteColor" size="1">
    <option>Red</option>
    <option>Green</option>
    <option>Blue</option>
 </select>
 </P>

<P><input type="submit" name="doit" value="Submit"/></P>
</form>
```

## CreateCookie

The `CreateCookie.xml` element sends a cookie named `ColorCookie` to the visitor's browser. If the visitor has disabled cookies, the browser ignores the request to set a cookie. If the visitor has enabled cookies (the default), the browser writes the cookie to this system's cookie file.

The following is the code for `CreateCookie.xml`:

```
<satellite.cookie NAME="ColorCookie"
VALUE="Variables.FavoriteColor"
TIMEOUT="31536000" SECURE="false"/>

<CALLELEMENT NAME="CSGuide/Sessions/DisplayWelcome"/>
```

The preceding code sets the value of the cookie to the visitor's favorite color. This cookie lasts for one year (31,536,000 seconds).

## DisplayWelcome

By the time DisplayWelcome is called, the cookie has been set. The following code uses the value of the cookie to display a welcome message in the visitor's favorite color.

```
<H1><font color="Variables.ColorCookie"
     REPLACEALL="Variables.ColorCookie">
```

```
Displaying a Friendly Welcome.
</font></H1>
```

### Running the Cookie Example

To run the cookie example, use your browser to go to the following pagename:

```
CSGuide/Sessions/Start
```

The first time you run this example, all four elements execute. After the first time, only `Start` and `DisplayWelcome` execute.

## Tips and Tricks

The following suggestions might be useful:

- In a cluster, session state must be replicated across cluster members. In a cluster, try to keep session size to a minimum; don't store more than 2 Kilobytes of session data per client.
- Determine reasonable session timeout values. Setting timeouts that are too large tie up system resources. Setting them too small forces visitors to log in with annoying frequency.

## Satellite Server Session Tracking

Web sites that present personalized content to visitors must track sessions. Content Server and Satellite Server both track sessions. Both set cookies in the visitor's browser; thus, two cookies (rather than one) each independently track a session. This redundancy is useful if a Satellite Server goes down; when a Satellite Server goes down, the Content Server session is maintained.

### Flushing Session Information

Though Satellite Server will only serve session-specific pagelets back to the person who originally requested them, explicitly flushing session-specific information on user logout is a wise way to conserve space in the Satellite Server cache.

The following sections describe how to flush session information from Satellite Server.

#### Flushing a Session Via URL

You can flush all data pertaining to a particular session. You do this from Content Server by posting a form to a URL in the following format:

```
https://host:port/servlet/
   FlushServer?reset=true&username=username&password=password&
   ssid=sessionID
```

where:

| Parameter | Value |
|---|---|
| host | Specify the name of the Satellite Server host whose cache is to be flushed |
| port | Specify 80 (the default) unless you reconfigured Resin to run on a different port. |
| username | Specify the value assigned to the username property. |
| password | Use the value assigned to the password property. |
| sessionID | Specify the session ID (the one maintained by Content Server, and not by Satellite Server) representing the session to be removed. |

## Flushing Current Session Information

To flush the information for the Satellite Server session that you are currently in, use the `FlushServer` URL with the current session's ID. The current session ID (`ssid`) is stored in a session variable with a name that is dependent upon your application server. You can see this name by looking at the session variable `HTTP_COOKIE`.

The following java code flushes the information for the current session:

```
String value;
String name = "WebLogicSession";
value = ics.GetVar(name);

String sFlushSessionUrl ="http://mysatellite:80/servlet/
   FlushServer?username=ftuser&password=ftuser&
   reset=true&ssid=" + value;"

String sSatTest1Results = Utilities.readURL(sFlushSessionUrl);
```

## Flushing Other Session Information

### To flush information from a session other than the one you are in

1. Add the following tag to the container page that contains the pagelets that you want to flush:

   ```
   <satellite.page
   pagename="QA/Satellite/Functional/xml/pagelet4"
   cachecontrol="session:0:00:00 */*/*"/>
   ```

   The `cachecontrol` value of `"session:0:00:00 */*/*"` means that every session that requests this page creates a pagelet that can only be viewed by subsequent requests by that session. Once the session for a given page expires, that page cannot be viewed again. The container page will expire from the cache at midnight each day.

2. After setting the cachecontrol parameter for the container page, use the Inventory servlet with the keys parameter to get its session ID (`ssid`). The `ssid` is the string that precedes the protocol and server name. For example, if the Inventory servlet displays:

```
OuCOTrh9yporWfgu8Uthttp://myserver:80/servlet/
ContentServer?pagename=QA/Satellite/Functional/xml/pagelet4
```

then the `ssid` is `OuCOTrh9yporWfgu8U`.

3. Flush information from the session by using the `ssid` you found with the FlushServer URL. For example:

```
http://myserver:80/servlet/
FlushServer?username=ftuser&password=ftuser&reset=true&
ssid=OuCOTrh9yporWfgu8U
```

**Note**

You should have session affinity enabled if you want to flush information from a session other than the one you are in.

Chapter 10

# Error Logging and Debugging

Content Server provides several options for logging error messages and debugging source code. This chapter gives you information about general error logging and debugging techniques that apply throughout the Content Server development environment. It contains the following sections:


- Overview
- Logging to Content Server's Log File
- Using Error Codes with Tags

# Overview

Content Server can log its activity in a log file, which by default is named `futuretense.txt`, typically located in the Content Server installation directory. For example:

```
ContentServerInstallDir/futuretense.txt
```

The type and volume of information that is written to the log file is controlled by the loggers that you choose to enable or define. See "Logging to Content Server's Log File" for information about loggers.

Content Server also has a reserved variable that is used by JSP and XML tags for returning an error code if the tag did not successfully complete its task. See "Using Error Codes with Tags" for more information.

# Logging to Content Server's Log File

Logging is based on either the commons-logging framework (using the `commons-logging.properties` file) or the Apache log4j framework.

As of version 7.6, log4j can be set up as Content Server's logging system either during the installation process or manually after installation. When log4j is set up, the `log4j.properties` file is created to specify how information must be logged and which information will be logged. In addition, the Content Server Advanced interface provides the **Configure log4j** tool in the **System Tools** node, on the Admin tab. Using **Configure log4**j, you can view current loggers directly from the Advanced interface. You can also dynamically add new loggers and change logger levels. Changes will persist upon system restart if you copy the text version of the loggers from the interface to the `log4j.properties` file. For more information about the **Configure log4j** tool, see the *Content Server Administrator's Guide*.

For information about the `log4j.properties` and `commons-logging.properties` files, see the *Content Server Property Files Reference.*

To define your own loggers or write your own messages to Content Server's log file, use the `ics:logmsg` tag. The following example writes a warning message to Content Server's log file.

```
<ics:logmsg msg="This is a warning message"
            name="com.fatwire.logging.cs.jsp"
            severity="warn"/>
```

For more information about `ics:logmsg`, see the *Content Server Tag Reference*.

**Note**

FatWire recommends that you set loggers to a level that agrees with the type of system on which logging is implemented. On development and content management systems, logging levels can be set to a greater severity (such as `INFO` or `DEBUG`), which provides a large amount of information. On delivery systems, loggers can be either disabled or set to low severity (`WARN` or `ERROR`) to avoid performance setbacks and making system information available on a publicly accessed environment.

# Using Error Codes with Tags

Content Server has a reserved variable named `Variables.errno` which most JSP and XML tags use for returning an error code (generally referred to as an **errno**) if the tag did not successfully complete its task.

For example, the `<CALLELEMENT>` XML tag sets `Variables.errno` as follows:

- `-10` if you specified a nonexistent element.
- `-12` if you specified an existing element that Content Server could not evaluate.

On success, `<CALLELEMENT>` does not modify the value of `Variables.errno`.

**Note**

For revision tracking operations, the reserved variable named `Variable.errdetails` provides additional information about the error.

You typically use the following strategy with tags that use `Variables.errno`:

1. Initialize `Variables.errno` to `0` before calling the tag.
2. Call the tag.
3. Evaluate `Variables.errno`.

## Tag Examples Using Error Codes

For example, the following code performs all three steps:

```
<SETVAR NAME="errno" VALUE="0"/>
<SETCOUNTER NAME="pi" VALUE="3.14159"/>
<IF COND="Variables.errno=-501">
 <THEN>
       <p>Bad value of pi</p>
 </THEN>
</IF>
```

Running this code yields the following HTML because `SETCOUNTER` cannot handle floating-point values:

```
<p>Bad value of pi</p>
```

The `ASSET`, `RENDER`, and `SITEPLAN` tags clear errno before they execute. You do not need to set errno to 0 when you use these tags. For example, after you use an `ASSET` tag, just check the value of errno to determine whether it has changed:

```
<ASSET.LOAD NAME="topArticle" TYPE="Article"
OBJECTID="Variables.cid"/>
<IF COND="IsError.Variables.errno=false">
      <THEN>
         <ASSET.CHILDREN NAME="topArticle"
            LIST="listOfChildren”/>
      </THEN>
</IF>
```

At the end of CS-Direct template elements, you can include error checking code such as this:

```
<IF rendermode="preview">
      <THEN>
         <IF COND="IsError.Variable.errno=true">

            <THEN>
            <FONT COLOR="#FF0000">
            Error <CSVAR NAME="Variables.errno"/>
            while rendering <CSVAR NAME="pagename"/>
            with asset ID <CSVAR NAME ="Variables.cid"/>.</
FONT>
            </THEN>
         </IF>
      </THEN>
</IF>
```

### Java Interface

After making calls to Content Server, the String variable `errno` can be retrieved and tested for success or failure. Here's an example:

```
cs.clearErrno();
IList rslt = cs.SelectTo(SYSTEMUSERS_TABLE, ALL_FIELDS,
USERNAME,
             null, NO_LIMIT, null, CACHE_RESULTS, errstr);
errno = cs.GetVar("errno");

if (errno.compareTo(ERRNO_SUCCESS) == 0)
    {
    ...
```

## Error Number Rules

Error numbers are always integers. The following table briefly summarizes error numbering rules for `Variables.errno`.

See the *Content Server Tag Reference* for specific error numbers for each tag.

| Number | Significance |
|---|---|
| Negative integers | Failure |
| 0 (zero) | Success |
| Positive integers in a tag other than a revision tracking tag. | Information |
| Positive integers in a revision tracking tag. | Failure |

Part 3

# Data Design

This part describes the Content Server database and explains how to design the data (assets) that your Content Server system delivers.

It contains the following chapters:


- Chapter 11, “Data Design: The Asset Models”
- Chapter 12, “The Content Server Database”
- Chapter 13, “Managing Data in Non-Asset Tables”
- Chapter 14, “Resultset Caching and Queries”
- Chapter 15, “Designing Basic Asset Types”
- Chapter 16, “Designing Flex Asset Types”
- Chapter 17, “Flex Filters”
- Chapter 18, “Designing Attribute Editors”
- Chapter 19, “Configuring Bundled Attribute Editors”
- Chapter 20, “Importing Assets of Any Type”
- Chapter 21, “Importing Flex Assets”
- Chapter 22, “Importing Flex Assets with the BulkLoader Utility”

Chapter 11

# Data Design: The Asset Models

The Content Server servlets are the operating system that runs your Content Server content management system—but the Content Server database is the brains of the system. It stores the system information that makes the Content Server applications run, the content that you are using the Content Server applications to manage (that is, assets), and the structural information that provides the format and business logic for displaying your content to the visitors of your online sites.

For the most part, data design means asset design. However, developers frequently need to create tables that hold supporting data for their assets. Determining the need for those tables and then designing them is also a part of data design.

This chapter contains the following sections:


- Asset Types and Asset Models
- The Basic Asset Model
- The Flex Asset Model
- Assetsets and Searchstates
- Search Engines and the Two Asset Models
- Tags and the Two Asset Models
- Summary: Basic and Flex Asset Models
- Summary: Asset Types


**Note**

Designing and creating tables that do not hold assets is discussed in Chapter 12, “The Content Server Database.”

# Asset Types and Asset Models

An **asset** is an object that is stored in the Content Server database, an object that can be created, edited, inspected, deleted, duplicated, placed into workflow, tracked through revision tracking, searched for, and published to your delivery (live) site.

An **asset type** is a definition or specification that determines the characteristics of asset objects of that type.

Developers design and create asset types while designing your content management system and your online sites. Content providers then create and edit assets of those types

In general, assets perform one of the following three roles:

- Provide **content** that visitors read and examine on your online sites
- Provide the **formatting** logic or code for displaying the content
- Provide **data structure** for storing the content in the Content Server database

The developer's job is to design asset types that are easy for content providers to work with on the management system and that can be delivered efficiently to visitors from the delivery system.

## Two Data Models

The Content Server products provide two data models for the assets types that you design: **basic** and **flex**.

- **Basic** asset types have a **simple data structure**: they have one primary storage table and simple parent-child relationships with each other.

  The basic asset model is delivered with CS-Direct.

  Basic asset types are separate, standalone asset types that represent individual kinds of content: an article, an image file, a page, a query, and so on. You use the AssetMaker utility (located on the **Admin** tab in the Content Server interface when CS-Direct is installed) to create new basic asset types.

- **Flex** asset types have a **complex data structure** with several database tables and the ability to support many more fields than do basic asset types. Additionally, they can have more than one parent, any number of grandparents, and so on, that they can **inherit** attribute values from.

  The flex asset model is delivered with CS-Direct Advantage only. You cannot create flex asset types with CS-Direct.

  Flex asset types comprise families of asset types that define each other and assign attribute values to each other. You use the Flex Family Maker utility (located on the **Admin** tab in addition to the AssetMaker utility when you have CS-Direct Advantage installed) to create a family of flex asset types.

## Default (Core) Asset Types

Several core asset types are delivered by CS modules and products. Because Content Server has a stack architecture, the core asset types are made available as follows:

- CS-Direct delivers the template, query, collection, SiteEntry, CSElement, Link, and page asset types. All of the other modules and products use the template and page asset types.

- CS-Direct Advantage delivers the attribute editor asset type. It supports any flex attribute asset types that you create.
- Engage delivers the visitor attribute, history attribute, history definition, segment, recommendation, and promotion asset types.

Assets of these types provide format or logic for the display of asset types that hold your content by retrieving, ordering, organizing, and formatting those assets. In other words, you use the core asset types to organize and format the content on your online site.

## CS-Direct

The core asset types delivered with CS-Direct provide basic site design logic. You can create as many individual assets of these types as you need, but you cannot modify the asset types themselves:

- **Query** stores queries that retrieve a list of assets based on selected parameters or criteria. You use query assets in page assets, collections, and recommendations. The database query can be either written directly in the “New” or “Edit” form for the query asset as a SQL query, or written in an element (with Content Server query tags or a as a search engine query) that is identified in the “New” or “Edit” form.
- **Collection** stores an ordered list of assets of one type. You “build” collections by running one or more queries, selecting items from their resultsets, and then ranking (ordering) the items that you selected. This ranked, ordered list is the collection. For example, you could rank a collection of articles about politics so that the article about last night’s election results is number one.
- **Page** stores references to other assets. Arranging and designing page assets is how you represent the organization or design of your site. You design page assets by selecting the appropriate collections, articles, imagefiles, queries, and so on for them. Then, you position your page assets on the **Site Plan** tab that represents your site in the tree on the left side of the Content Server interface.

  Note that a page asset and a Content Server page are quite different. The page asset is an organizational construct that you use in the Site Plan tab as a site design aid and that you use to identify data in your elements. A Content Server page is a rendered page that is displayed in a browser or by some other mechanism.
- **Template** stores code (XML or JSP and Java) that renders other assets into Content Server pages and pagelets. Developers code a standard set of templates for each asset type (other than CSElement and SiteEntry) so that all assets of the same type are formatted in the same way.

  Content providers can select templates for previewing their content assets without having access to the code itself or being required to code.
- **CSElement** stores code (XML or JSP and Java) that does not render assets. Typically, you use CSElements for common code that you want to call from more than one template (a banner perhaps). You also use CSElements to provide the queries that are needed to create DynamicList recommendations in Engage.
- **SiteEntry** represents a Content Server page or pagelet and has a CSElement assigned as the root element that generates the page. Template assets do not have associated SiteEntry assets because they represent both an element and a Content Server page.
- **Link** stores a URL to an external web site. You use this asset to embed an external link within another asset.

Because the data needs of each organization using a Content Server content management system are different, there are no default asset types that represent content. However, the sample sites deliver sample content asset types that you can examine and modify for use on your sites.

## CS-Direct Advantage

There is one core asset type in the CS-Direct Advantage application: **attribute editor.**

An attribute editor specifies how data is entered for a flex attribute when that attribute is displayed on a “New” or “Edit” form for a flex asset or a flex parent asset. It is similar to a Template asset. However, unlike a Template asset, you use it to identify the code that you want CS-Direct Advantage to use when it displays an attribute in the Content Server interface—not when it displays the value of an attribute on your online site.

## Engage

The Engage application delivers several core asset types that you use to gather visitor information so that you can personalize the product placements and promotional offerings that are displayed for each visitor:

- **Visitor attribute** holds types of information that specify one characteristic only (scalar values). For example, you can create visitor attributes named “years of experience,” “job title,” or “number of children.”
- **History attributes** are individual information types that you group together to create a vector of information that Engage treats as a single record. This vector of data is the **history definition**. For example, a history type called “purchases” can consist of the history attributes “SKU,” “itemname,” “quantity,” and “price.”
- **Segments** are assets that divide visitors into groups based on common characteristics (visitor attributes and history types). You build segments by determining which visitor data assets to base them on and then setting qualifying values for those criteria. For example, a segment could define people who live in Alaska and own fly fishing gear, or it could define people who bought a personal computer in the past six months, and so on.

After you define and categorize the visitor data that you want to collect, you use the following asset types to select, organize, and display the flex assets that represent your content on your online site:

- **Recommendation** is something like an advanced collection. It collects, assesses, and sorts flex assets (products or articles, perhaps) and then recommends the most appropriate ones for the current visitor, based on the segments that visitor belongs to.
- **Promotion** is a merchandising asset that offers some type of value or discount to your site visitors based on the flex assets (products, perhaps) that the visitor is buying and the segments that the visitor qualifies for.

**Note**

Engage interacts with assets that are built using the **flex asset model only**. You cannot program recommendations and promotions to work with assets that use the basic asset model.

## Which Asset Model Should You Use to Represent Your Content?

During the process of designing your online site with the Content Server content management system, you and others on your team create the asset types that you need to represent the content for your site. The CS-Direct template and page asset types provide the formatting framework for the asset types that represent your data, whether you use the basic data model or the flex data model.

The asset data model (basic or flex) that you should choose to represent the data that you want to display on your online site depends on the nature of that data, as described in the following two sections.

### When to Use the Basic Model

The basic model is a good choice when your data has the following characteristics:

- It is fixed, predictable: there will be no need to add attributes to the asset type.
- It is homogenous: all assets of the same type have similar attributes.
- It has a moderate number of attributes. You are limited by your database as to how many columns/attributes you can have in the asset type table for a basic asset.
- You want to use the static publishing method. There are very limited applications of the flex asset model in which it makes sense to use the static publishing method.
- Visitors browse your online site by navigating from link to link.

When the data for an asset type can be imagined as a spreadsheet, as a simple flat table where each asset of that type is a single record and every record has the same columns, that asset type should use the basic asset model.

### When to Use the Flex Model

The flex model is the right choice when your data has the following characteristics:

- It has lots of attributes. For example, products can have potentially hundreds of attributes. Because attribute values for the flex family member are stored as rows rather than columns, flex assets can physically have many more attributes than basic assets can.
- It can be represented in a hierarchy in which assets inherit attribute values from parent assets.
- You cannot predict what attributes might be necessary in the future and your data might need additional attributes periodically.
- Asset instances of the same type can vary widely. That is, not all assets of that type should have the same attributes. For example, a bath towel product asset would have attributes that a toaster product asset would not, but both the bath towel and the toaster are product assets.
- Visitors browse your online site by navigating through “drill-down” searches that are based on the attribute values of your data.
- You want to use Engage.

For example, products fit into the flex asset model because markets are constantly changing. You cannot always predict what products you will be selling next year or what attributes those products will have.

If your business needs will require you to make modifications to your asset types such as adding or changing their attributes, the flex data model is probably the right choice for you. The flex asset model gives you the extensibility that you need to represent data whose characteristics cannot be predicted.

# The Basic Asset Model

CS-Direct delivers the basic asset model. In general, the data model for basic asset types is one database table per asset type. All basic assets of the same type have the exact same fields (properties) and all assets of a single type are stored in the same database table.

Most of the core CS-Direct asset types use the basic data model.

To create new basic asset types, you use the AssetMaker utility. You code XML files called **asset descriptor files** using a custom tag named `PROPERTY` and then upload the file with AssetMaker. A **property** is both a column and a field. A `PROPERTY` statement defines a column in the table that stores assets of that type and defines how data is to be entered into the corresponding field for that column in the CS-Direct forms.

For information about coding asset descriptor files and creating new basic asset types, see Chapter 15, “Designing Basic Asset Types.”

## Basic Asset Types from the Burlington Financial Sample Site

If you installed the Burlington Financial sample site, CS-Direct installed five asset types that represent content. These sample site asset types use the basic asset model delivered with CS-Direct:

- **Article** stores the text of an article and information about it. It has fields for headline, byline, credit line, body, and so on. Note that this is a custom asset type that was **not created with AssetMaker.**
- **ImageFile** stores an image file as an uploaded binary large object (blob). These image files can be associated with other assets such as a page or an article. This asset type was created with AssetMaker.

This sample site also provides an example of how you can create additional formatting asset types, if necessary, with the following asset type:

- **StyleSheet** stores style sheet files of any format (CSS, XSL, and so on). You create the style sheet in a text editor and then upload it into CS-Direct as a style sheet asset. When you store style sheets as assets, you can assign a workflow to them, use revision tracking, and so on. This asset type was created with AssetMaker.

Burlington Financial installs the following asset types, but does not use them:

- **Linkset** stores a group of links to either the URLs of related assets or the URLs of external Web sites. Assets of this type can be associated with other assets like a page or an article.
- **Image** stores the URL for an image file that can be associated with other assets like a page or an article.

These asset types were used by a previous sample site. They are included for backward compatibility.

## Relationships Between Basic Assets

Basic asset types have very simple parent-child relationships. You use these relationships to associate or link assets to each other. Then, when you design the online pages for your online sites you code template elements that identify, extract, and then display an asset's children or parent assets in appropriate ways.

The relationships that basic assets can have with each other are called **named associations** and **unnamed relationships**. When these relationships occur between individual assets, they are written to the `AssetRelationTree` table.

### Named Associations

Named associations are defined, asset-type-specific relationships that are represented as fields in the CS-Direct asset forms. After you create an asset type with AssetMaker, you use the "Association" form for that asset type to create association fields.

You use named associations to set up relationships that make sense for the asset types in your system and then you use the names of these relationships to identify the related assets and display them in appropriate ways on your site pages.

For example, the Burlington Financial sample asset named article has three named associations with the imagefile asset type: Main ImageFile, Teaser ImageFile, and SpotImageFile. The Burlington Financial article templates are coded to display the imagefiles that are linked to articles through these associations. The association is what enables the template to determine which imagefile is the correct one to display for an individual article asset.

When a content provider selects an image asset in the **Main Image** field of the "New" and "Edit" article forms, the selected imagefile asset becomes a child of the article asset. (Note that this same imagefile asset can also be a child of other articles.)

When you create a new named association between asset types, CS-Direct creates a row for that type of association in the `Association` table. Then, when you create an asset and specify the name of another asset in an association field, that relationship is written to the `AssetRelationTree` table.

### Unnamed Relationships

Unnamed relationships occur in the following situations:

- When you build a collection, the items in the collection become children of the collection.
- When you select queries or other assets for page assets from the tree, which places them in the **Contains** list box, on the "New" and "Edit" forms for page assets, those assets become children of the page asset.

Neither of these relationships is identified by a name.

The Burlington Financial sample site has two page assets with an unnamed relationship set up through that page asset's **Candidates** list. The "About Us" article placed on the About page asset has an unnamed relationship to the About page asset. The article is a child of the page asset. Additionally, the "Contact Us" article placed on the Contact page asset has an unnamed relationship to the Contact page asset.

Because there is no name for these kinds of relationships, CS-Direct does not create rows in the `Association` table for them. However, the individual instances of these unnamed associations are written to the `AssetRelationTree` table.

## Category, Source, and Subtype

There are three additional ways to organize or categorize basic assets: category, source, and subtype. Categories and subtypes are specific to an asset type. Source, however, applies to all the asset types in a content management site. In other words, source is site-specific.

### Category

`Category` is a default column and field that you can use to categorize assets according to a convention that works for your sites. Although all basic asset types have a `category` column by default, you do not have to use it (it is not a required field).

For example, the Burlington Financial sample site has categories named Personal Finance, Banking and Loans, Rates and Bonds, News, and so on. Articles identified with these categories are selected by queries that use "category" as a selection criterion and displayed on specific site pages, as appropriate.

When you create a new basic asset type, AssetMaker creates one category code for assets of that type. You then use the "Category" form for your new asset type to create additional categories if you want to use this feature.

New categories are written to the `Category` table, which serves as the lookup table for the **Category** field on the "New" and "Edit" asset forms for asset types that use the basic asset model.

The purpose of the **Category** field and column is for site design. You can use category, or not, in your queries and query assets for your online site. The CS-Direct application does not base any of its functions on category codes. (With the exception that you can Search for assets based on this field, if you are using it.)

### Source

`Source` is a column and field that you can use to identify where an asset originated. Although CS-Direct provides administrative support (through the "Source" form) for you to use this feature in the design of your online site, the `source` column does not exist by default in the primary storage tables for basic asset types other than Article. If you want to use source with your basic asset types, you must include a property statement in your asset descriptor file for it.

For example, the Burlington Financial sample site has sources named WireFeed, Asia Pulse, UPI, and so on. Certain online pages select stories to display based on the results of queries that search for articles based on the value in their source column.

After you create a new basic asset type, you add new sources in the "Source" form on the **Admin** tab, if necessary. New sources are written to the `Source` table, which serves as the lookup table for the **Source** field on the "New" and "Edit" asset forms for basic-style assets.

### Subtype

The **subtype** concept provides a way to further classify an asset type. In the flex asset data model, the definition asset types create subtypes of flex assets and flex parent assets. In the basic asset data model, the concept of subtype is implemented through the `subtype` column in the primary storage table for the asset type.

The CS-Direct application uses the value of an asset's **Subtype** in many ways:

- For Template assets, subtype means the type of asset that the template formats. Templates that format articles are a different subtype of template than templates that format images. When you create an article asset, only the templates that format articles appear as options in the **Template** field on that asset's "New" or "Edit" form.

  In addition, you can use the Content Server user interface to specify a subtype that will be displayed using a given template. For example, if your web site uses two subtypes of article asset, Sports and News, you can create a template that only displays articles with the Sports subtype.
- For query assets, subtype means the type of asset that the query returns. Query assets that return articles are a different subtype of query asset than those that return imagefiles.
- For collection assets, subtype means the type of asset that the collection holds. Collections that hold articles are a different subtype of collection asset than those that hold imagefiles.
- For the basic asset types that you design, subtype is designed to classify an asset based on how it is rendered. You can define a default template for each subtype of an asset type for each of your publishing targets.

If you do not need to assign a different template to assets of a specific type based on the publishing target for the asset, you do not need to create new subtypes.

If you create any subtypes for an asset type, the "New" and "Edit" forms for assets of that type display a field named **Subtype**. The drop-down list in the field displays all the possible subtypes for that asset type.

**Note**

In the flex asset model, the definition asset types serve as subtypes. For example, in the GE Lighting sample site, there is one product definition: lighting. This means that there is one subtype for product assets: the lighting subtype.

For some asset types, the subtype is set implicitly and cannot be changed. Other asset types allow users to choose a subtype for the asset using the Content Server user interface. The following table lists the Content Server asset types according to whether they have configurable subtypes:

| Implicit Subtypes | Configurable Subtypes |
|---|---|
| • All flex assets<br>• Query assets<br>• Collection assets<br>• Template assets | • All custom basic assets (made with AssetMaker)<br>• Article assets<br>• Image assets<br>• Linkset assets<br>• Recommendation assets<br>• Link assets<br>• Page assets |

For information about setting configurable subtypes, see Chapter 15, "Designing Basic Asset Types."

## Basic Asset Types and the Database

Although there is one primary storage table for basic asset types, CS-Direct keeps other kinds of supporting information for basic assets in other tables. When you create a new asset of a basic type, CS-Direct writes to the following database tables:

- The primary database table that holds assets of its type. For example, each page asset has a row in the `Page` table and each article asset has a row in the `Article` table.

  These tables store all of the asset's attribute or field values, such as the asset's name, its object ID, who created it, which template it uses, and so on. The name of this table always matches the name of the asset type.

  When you create a new basic asset type, the AssetMaker utility creates the primary storage table (a Content Server object table) for the asset type as a part of that process.
- The `AssetRelationTree` table, if the asset has unnamed parent-child relationships or named associations with other assets. (The relationships that basic assets can have are described in "Relationships Between Basic Assets" on page 187.)
- The `AssetPublication` table, which specifies which content management sites (publications) give you access to the asset. If the asset is shared among sites (publications), there is a row entry for each pubid. A **pubid** is a unique value that identifies a site (publication).
- The `SitePlanTree` table, if the asset is a page asset. This table stores information about the page asset's hierarchical position in your site plan.

When you develop the templates that display the assets that represent your content, you code elements with CS-Direct XML or JSP tags that extract and display the information from the tables in the preceding list.

Be sure to examine the CS-Direct "New" and "Edit" forms for the various sample asset types and to use the Content Server Explorer tool to examine the tables in your Content Server database.

**Note**

Do **not** use Content Server Explorer tool to modify the data in any of these tables. All editing of assets and their related tables should be done only through the Content Server interface.

## Template Asset Type and the Database

Although the Template asset type is a core CS-Direct asset type, it does not use the basic asset model. It is a complex asset type with entries in the following database tables:

- The `Template` table, its primary storage table
- The `SiteCatalog` table
- The `ElementCatalog` tables

When you create a new Template asset, CS-Direct automatically creates entries in both the `SiteCatalog` and `ElementCatalog` tables for it. For more information about Template assets, see Chapter 25, "Creating Collection, Query, Stylesheet, and Page Assets."

## Default Columns in the Basic Asset Type Database Table

CS-Direct needs several default columns for its basic functionality and so AssetMaker creates each of the following columns (as shown in the following table) in the asset type's primary storage table in addition to the columns defined in the asset descriptor file for that asset type.

Note that you do not need to code your asset descriptor files to include property statements for the columns in this list:

**Table 4:** Columns in an Asset Type's Primary Storage Table

| Default Column (Field) Name | Description | Where It's Displayed in the Content Server Interface |
|---|---|---|
| `id` | A unique ID for each asset, automatically generated by Content Server when you create the asset.<br>You cannot change the value in this field. | Forms:<br>• Inspect<br>• Edit<br>• Status<br>• search forms |
| `name` | A unique name for the asset. Names are limited to 64 alphanumeric characters. | Forms:<br>• New<br>• Edit<br>• Inspect,<br>• Status<br>Also in the search results lists. |
| `description` | A short description of the asset that offers more information than just the name. | Forms:<br>• New<br>• Edit<br>• Inspect<br>• Status<br>Also in the search results lists. |

**Table 4:** Columns in an Asset Type's Primary Storage Table *(continued)*

| Default Column (Field) Name | Description | Where It's Displayed in the Content Server Interface |
|---|---|---|
| `status` | The status of the asset, one of the following status codes obtained from the `StatusCode` table:<br>PL - created<br>ED - edited<br>RF - received (from XMLPost, for example)<br>UP - upgraded from Xcelerate 2.x<br>VO - deleted (void)<br>CS-Direct controls the value in this field; it cannot be edited manually. | Forms: **Status**, if the status of an asset is either PL (created) or ED (edited)<br>Note that assets with a status of VO (deleted) are not displayed anywhere in the Content Server Windows interface. |
| `createdby` | The identity of the user who originally created the asset. This user name is obtained from the `SystemUsers` table.<br>CS-Direct controls the value in this field; it cannot be edited manually. | Forms: **Status**<br>Also, if revision tracking is enabled for assets of this type, the **Revision History** list. |
| `createddate` | The date and time that the asset was written to the database for the first time.<br>CS-Direct controls the value in this field; it cannot be edited manually. | Forms: **Status**<br>Also, if revision tracking is enabled for assets of this type, the **Revision History** list. |
| `updatedby` | The identity of the user who most recently modified the asset in any way. This user name is obtained from the SystemUsers table.<br>CS-Direct controls the value in this field; it cannot be edited manually. | Forms: **Status**<br>Also, if revision tracking is enabled for assets of this type, the **Revision History** list. |
| `updateddate` | The date on which the information in the status field was changed to its current state.<br>CS-Direct controls the value in this field; it cannot be edited manually. | Forms: **Status**<br>Also, if revision tracking is enabled for assets of this type, the **Revision History** list. |

**Table 4:** Columns in an Asset Type's Primary Storage Table *(continued)*

| Default Column (Field) Name | Description | Where It's Displayed in the Content Server Interface |
|---|---|---|
| `startdate` | Promotion assets (a Engage asset) have durations during which they can be displayed on the visitor pages on your live system. This column stores the start time of the promotion's duration.<br>The promotion asset type is the only default asset type that uses this column.<br>If you want to use the startdate and enddate fields for your asset types, see "Example: Enabling path, filename, startdate, and enddate" on page 285. | Forms:<br>• **Duration**, **Edit**, and **Inspect** for promotion assets.<br>• **New**, **Edit**, **Inspect**, and **Status** if you enable it for other asset types. |
| `enddate` | For promotion assets (a Engage asset), this column stores the end time of the promotion's duration<br>The promotion asset type is the only default asset type that uses this column. | Forms:<br>• **Duration**, **Edit**, and **Inspect** for promotion assets<br>• **New**, **Edit**, **Inspect**, and **Status** if you enable it for other asset types |
| `subtype` | The value of the asset's subtype. The subtype is set in different ways for different assets. For more information, see "Subtype" on page 188. | Forms:<br>• **New**, and **Edit** for Template assets (**Asset Type** field)<br>• **New**, and **Edit** for query assets (**Result of Query** field)<br>• **New**, and **Edit** for any asset type that has subtypes configured for it<br>• **Set Default Templates** |

**Table 4:** Columns in an Asset Type's Primary Storage Table *(continued)*

| Default Column (Field) Name | Description | Where It's Displayed in the Content Server Interface |
|---|---|---|
| `filename` | The name to use for the file created for this asset during the Export to Disk publishing method.<br>The page and article asset types are the only asset types that have this field enabled by default.<br>If you want to use the filename field for your asset types, see "Example: Enabling path, filename, startdate, and enddate" on page 285 | Forms:<br>• **New** and **Edit** for page and article assets, by default<br>• **New** and **Edit** for any other asset type that has the field enabled |
| `path` | The directory path to use for exported page files that are generated from child assets of this asset when the Export to Disk publishing method renders that asset into a file.<br>The page and article asset types are the only asset types that have this field enabled by default.<br>If you want to use the filename field for your asset types, see "Example: Enabling path, filename, startdate, and enddate" on page 285. | Forms:<br>• **New** and **Edit** for page and article assets, by default<br>• **New** and **Edit** for any other asset type that has the field enabled |
| `template` | The template for the asset.<br>This is the template that is used to render the asset when it is either published with Export to Disk or rendered on a live dynamic delivery system.<br>This template is also used to calculate the dependencies when the asset is approved for the Export to Disk publishing method, unless the asset type has subtypes and there is a default approval template assigned for the asset based on its subtype. | Forms:<br>• **New**<br>• **Edit**<br>• **Inspect**<br>• **Status** |
| `category` | The category code of the category assigned to the asset, if any.<br>If you decide to use the category field to organize assets, you add category codes in the "Asset Types" forms on the **Admin** tab. | Forms:<br>• **New**<br>• **Edit**<br>• **Inspect**<br>• **Status** |

**Table 4:** Columns in an Asset Type's Primary Storage Table *(continued)*

| Default Column (Field) Name | Description | Where It's Displayed in the Content Server Interface |
|---|---|---|
| `urlexternaldoc` | If the asset was entered with the CS-Desktop interface rather than the Content Server interface, stores the external document that is the source for the asset.<br>CS-Direct controls the value in this field; it cannot be edited manually. | not applicable |
| `externaldoctype` | The mimetype of the file held in the `urlexternaldoc` field.<br>CS-Direct controls the value in this field; it cannot be edited manually. | not applicable |
| `urlexternaldocxml` | Reserved for future use. | not applicable |

# The Flex Asset Model

The flex asset model has the following main characteristics:

- Flex assets are defined by flex definitions–an asset type that determines which flex attributes make up an individual flex asset. Flex definitions create subtypes of the flex asset type.
- The definition asset types create subtypes of flex and flex parent assets, which allows individual instances of a flex asset or flex parent asset type to vary widely.
- Flex attributes are assets. The flex data model allows you to add flex attributes to (or remove them from) existing flex asset types at any time.
- Flex filters can take the data from one flex attribute, transform or assess it in some way, and then store the results in another flex attribute when you save the flex asset. The resulting value from a flex filter action is called a "derived" attribute value. For more information about flex filters, see Chapter 17, "Flex Filters."
- Flex assets can inherit attribute values—even derived values—from their flex parents, which means that you can represent your data in hierarchies.

You do not create individual flex asset types as you do basic asset types; instead, you create a flex family of asset types.

## The Flex Family

The flex asset data model can be thought of in terms of a family of asset types. There are six asset types in a flex family. Five are required, the sixth is optional, as indicated in the table below.

| Flex Family Member | Number Per Family |
|---|---|
| flex attribute asset type | one |
| flex parent definition asset type | one or more |
| flex definition asset type | one or more |
| flex parent asset type | one or more |
| flex asset type | one or more |
| flex filter asset type | none or more |

Whereas some of the asset types are used exclusively by developers to create the other asset types in the data model, the flex asset type is always used by the content providers to create assets of that type. (When necessary, authorized users can be given access to additional flex family members.)

To create a flex family, you use the “Flex Family Maker” forms on the **Admin** tab in the Content Server interface. You name each of the asset types in the family. For example, one of the flex families in the GE Lighting sample site is the product family. The flex asset is called the product asset, the flex attribute is called the product attribute, and so on.

The key member of a flex family is the **flex asset**. The flex asset is the unit of data that you extract from the database and display to the visitors of your online site (delivery system). All of the other members in the family contribute to the flex asset member in some way.

While the flex asset is the key, the **attributes** are the foundation of the flex asset model. An attribute is an individual component of information. For example, color, height, author, headline. You use attributes to define the flex assets and the **flex parents**. Flex assets inherit attribute values from their parents who inherit attribute values from their parents and so on.

You decide which attributes describe which flex assets and which flex parents by creating “templates” with the **flex definition** and **flex parent definition** asset types. Flex parents and their definitions implement the inheritance of attribute values.

Note that a flex parent or a flex asset cannot be defined by attributes of two types. The GE sample site has two kinds of attributes: product attributes and content attributes. A product asset (the flex asset member in the product flex family) can be defined by product attributes only—its definition cannot include content attributes.

A **flex filter** enables you to configure some kind of action to take place on the value of an attribute and then save the results of the action when the flex asset is saved. For example, you can configure a filter that converts the text in a Word file into HTML code.

In summary, the flex asset member of a flex family is the reason for the family, the unit of content that you want to display. The other members of a flex family provide data structure for the flex asset. However, because all of the members in the family are assets, you can take advantage of the standard CS-Direct features like revision tracking, workflow, search, and so on.

## Parent, Child, and Flex Assets

When you are using the flex asset data model, the phrase "parent-child" relationship refers to the relationship between a flex asset and its flex parent asset(s). This is a different parent-child relationship than the ones that basic assets have through named associations and unnamed relationships.

Although it is possible for flex assets to have the kinds of parent-child relationships that basic assets do, it is unlikely for the following reasons:

- CS-Direct Advantage provides the `ASSETSET` and `SEARCHSTATE` tag families, which you use instead of the collection and query asset types to select the flex assets that you want to display. For more information about this tag family, see "Assetsets and Searchstates" on page 206.
- Flex assets have no need for named associations. For example, if you want to assign an image file to a flex asset like a product, you can create an attribute that identifies the image file and assign it to the definition for the flex asset.
- While assets that are selected from the **Candidates** list box on a page asset have an unnamed parent-child relationship with that page asset, when you are using the flex asset model, it is unlikely that you would place a flex asset directly onto a page asset.

# Sample Site Flex Families

If the GE sample site is installed on your development system, there are two flex asset families that you can examine: the product family and the content family.

To better understand the following descriptions of the sample flex asset types, examine some of the product, article, and image assets in the Content Server interface as you read this section.

## The Product Family

The product family provides the data structure for the lighting products that are sold from the GE Lighting sample site. It creates an online catalog of lighting products.

These are the asset types in the product family:

- **Product attribute** is the flex attribute asset type used to define the products and product parents in the GE Lighting sample catalog. For example, there are product attributes named wattage, voltage, bulb size, ballast type, and so on.
- **Product** is the flex asset member of the product family. Product assets represent the lighting products that are sold from the GE Lighting sample site. In this online catalog, product names are numbers similar to a SKU number.
- **Product definition** is the flex definition asset type in the product family. It is used to create one subtype of products: lighting. The lighting definition formats (defines) all of the light bulbs in this online catalog.
- **Product parent** is the flex parent asset type in the product family. Product parents represent categories of products such as Compact Fluorescent, Fluorescent, Halogen, and so on.
- **Product parent definition** is the flex parent definition asset type in the product family. It is used to create subtypes of product parents. There are two: Category and Subcategory.

The product attribute, product definition, and product parent definition assets are listed on the **Design** tab because you use them for data design. The product and product parent assets are located on the sample site's **Product** tab.

The product asset is the reason for the product family: the GE Lighting sample site sells products.

### The Content Family

The content family provides the data structure for the articles that describe and images that illustrate the products that are sold from the GE Lighting sample site.

This is the content family:

- **Content attribute** is the flex attribute asset type used to define the articles (flex) and images (flex) that illustrate the products sold from the GE Lighting sample site.
- **Article (flex)** is a flex asset type that stores the text of an article and information about it. It has attributes such as byline, headline, subheadline, body, and so on.
- **Image (flex)** is a flex asset type that stores the URL of an image file. Although the GE Lighting sample site makes this asset type available, it does not use it.
- **Content definition** is the flex definition asset type in the content family. It is used to create one subtype of the article (flex) asset type called "story."
- **Content parent** is the flex parent asset type in the content family. Although the GE Lighting sample site makes this asset type available, it does not use it.
- **Content parent definition** is the flex parent definition asset type in the content family. Although the GE Lighting sample site makes this asset type available, it does not use it.

Notice that there are two flex asset types in the GE sample site's content family. They share attributes, parents, definitions, and parent definitions.

The content attribute, content definition, and content parent definition assets are listed on the **Design** tab because you use them for data design. The image (flex), article (flex), and content parent assets are located on the sample site's **Content** tab.

## Flex Attributes

**Flex attributes** are the foundation of the flex asset model. An attribute represents one unit of information. You use attribute assets to define flex assets and flex parents. They are then displayed as fields in the "New" and "Edit" forms for your flex assets and their parents.

An attribute is similar to a property for a basic asset. As does a property, an attribute defines the kind of data that can be stored in a column in a Content Server database table and describes a field in the CS-Direct Advantage forms. However, while a property defines one column in an asset type's database table, an attribute is an asset with database tables of its own.

This data structure (attributes as assets rather than columns) is a one of the main reasons why flex assets are so flexible.

Once again, a flex parent or a flex asset cannot be defined by attributes of two types. For example, the GE Lighting sample site product asset can be defined by product attributes only—its definition cannot include content attributes.

**Note**

If you remove a flex attribute from a flex definition, be aware that the data stored by those attributes is not deleted from the database (queries will continue to return this data).

## Data Types for Attributes

The data types for your attributes are defined by the Content Server database properties located in the `futuretense.ini file`, with the exception of the `money` data type, which is defined by a property in the `gator.ini` file (which is the name of the `.ini` file for CS-Direct Advantage).

Table 5 lists the data types for flex attributes, the properties that define the data types, and the files where the properties are located:

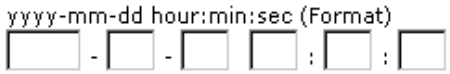
**Table 5:** Data types for flex attributes

| Type | Property | .ini file |
|---|---|---|
| date | cc.datetime | futuretense.ini |
| float | cc.double | futuretense.ini |
| integer | cc.integer | futuretense.ini |
| money | cc.money | gator.ini |
| string | cc.varchar | futuretense.ini |
| text | cc.bigtext | futuretense.ini |
| asset | cc.bigint | futuretense.ini |
| blob | cc.bigint | futuretense.ini |
| url (deprecated in version 4.0) | cc.varchar | futuretense.ini |

## Default Input Styles for Attributes

When a flex attribute is displayed as a field on a “New” or “Edit” form, it has default input styles based on its data types. The following list presents the default input styles for flex attributes:

- Date: input boxes that look like this:




- Float: text field with decimal position enforced.
- Integer: text field.

- Money: text field with currency format enforced.
- String: text field that accepts up to 255 characters.
- Text: text box. The number of characters that it accepts depends on the database and database driver you are using.
- Asset: drop-down list of all the assets of the type that was specified.
- Blob: a text field with a **Browse** button.
- URL: deprecated in version 4.0 but present for backward compatibility. You should use blob rather than URL.

If you do not want to use the default input style for a flex attribute, you can create an **attribute editor** and assign it to the attribute. Attribute editors are assets but they are also similar to the `INPUTFORM` statement in an asset descriptor file for a basic asset: they specify how data is entered into the attribute field. For more information about attribute editors, see Chapter 18, "Designing Attribute Editors."

### Foreign Attributes

You can have flex attributes that are stored in foreign tables, that is, foreign attributes. They are subject to the following constraints:

- The foreign table must be registered with Content Server. That is, the foreign table must be identified to Content Server in the `SystemInfo` table. For information, see "Registering a Foreign Table" on page 225.
- The foreign table must have a column that holds an identifier that uniquely identifies each row. The identifier must have fewer than 20 characters.
- The foreign table must have a column that is reserved for the attribute data value, which can be of any appropriate data type. For example, if the attribute is of type string, the data type must be appropriate for a string.

## Flex Parents and Flex Parent Definitions

**Flex parents** and their **flex parent definitions** are organizational constructs that do two things:

- Implement the inheritance of attribute values. The parent definitions set up (describe) the rules of inheritance and the parents pass on attribute values to the flex assets according to those rules of inheritance.
- Determine the position of a flex asset on the tabs that display your assets in the Content Server interface. The hierarchy of the parents and the flex assets on the tabs in that tree are based on the hierarchy set up with the parent definitions.

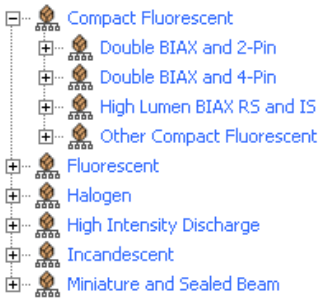
Each parent asset type has its own set of attributes, as specified in its parent definition. The parent definition creates a form that you see in the Content Server interface.

You use parents to organize or manage the flex assets by passing on attribute values that are standard and do not need to vary for each individual child asset of that parent.

Parent asset types affect how you and the content providers see and interact with the data within the Content Server interface.

For example, in the GE Lighting sample site there are two parent definitions: Category and SubCategory. Their sole purpose is to create structure on the sample site's **Product** tab in the tree (in the Content Server interface).

In the GE Lighting site, when the product parent's definition is Category, the product parent is displayed at the top level on the **Product** tab. When the product parent's definition is SubCategory, the product parent is displayed at the second level and it has a parent of its own:




For example, in the GE Sample site, there are several top-level product parents: Compact Fluorescent, Halogen, and so on. They were created with the Category definition. The next-level product parents, such as Double BIAX and 2-Pin, Double BIAX and 4-pin, and so on were created with the Subcategory definition.

## Business Rules and Taxonomy

The purpose of parent definitions and parent assets is not only to express the taxonomy of your data; they also allow you to apply business rules (logic) without risk of input error from end users. If, by creating a flex asset of a specific definition, there are dependencies that it should inherit, that flex asset should have a parent.

For example, here is a simple product, a toaster with five attributes:

- SKU = 1234
- Description = toaster
- Price = 20
- CAT1 = Kitchen
- CAT2 = Appliances

When the value of CAT2 is "Appliances," the value of CAT1 can only be "Kitchen." In other words, there is a business rule dependency between the value of CAT1 and the value of CAT2.

In this kind of case, there is no reason to require the content providers to fill in both fields. Because every field whose data has to be entered manually is a field that might hold bad data through input error, you would use inheritance to impose the business rule:

- Make CAT1 and CAT2 parent definitions.
- Make Kitchen a parent created with the CAT1 definition and Appliances a parent created with the CAT2 definition.
- Make Kitchen the flex parent of Appliances.

Now, when content providers create products, if they select Appliances for CAT2, the value for CAT1 is determined automatically through inheritance.

## Flex Assets and Flex Definition Assets

A **flex asset** is the reason for the flex family. It is the asset type that represents the end goal — a product, a piece of content that is displayed, and so on. For example, in the GE sample site there are three flex asset types:

- Product, which represents an individual saleable unit
- Article (flex), an asset that holds text
- Image (flex), an asset that holds the URL of a picture file

All of the other members in the family contribute to the flex asset member in some way.

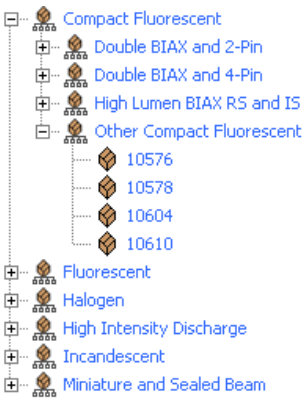
A **flex definition asset** describes one kind of flex asset in a flex family; for example, a shoe, a toaster, a bowling ball, a brochure, a newsletter, an article, and so on. A flex definition asset is a template in that it directly affects a form that you see in the Content Server interface.

Although the GE sample site has only one flex definition for products (lighting) and one flex definition for articles and images, you can create as many flex definitions as you need.

For example, if you were designing a product catalog that offered both toasters and linens, you would certainly create a flex definition asset for toasters and a different flex definition asset for linens.

Individual flex assets can be created according to only one flex definition asset. You could not create a product that used both the toaster definition and the linens definition.

A flex asset has not only the attributes assigned directly to it when it was created, it also has the attributes that it inherits from a parent. It can have more than one flex parent and whether the parents have parents depends on the hierarchical structure that you design. The products in the GE sample site, for example, have three levels of hierarchy:




The Other Compact Fluorescent product parent has a parent of its own (Compact Fluorescent) and several children (10576, 10578, and so on).

## Flex Families and the Database

Each asset type in a flex family has several database tables. For example, the flex asset member has six tables and a flex parent type has five. This data model enables the flex member in a flex family to support more fields than an asset type in the basic asset model can support.

The four most important types of tables in the flex model are as follows:

- The primary table for the asset type
- The `_Mungo` table, which holds attribute values for flex assets and flex parent assets only
- The `MungoBlobs` table, which holds the values of all the flex attributes of type blob.
- The `_AMap` table, which holds information about the inheritance of attribute values for flex asset and flex parents only

There are several other tables that store supporting data about the relationships between the flex assets as well as additional configuration information (details about search engines, the location of foreign attributes, publishing information, and, if revision tracking is enabled, version information).

Additionally, certain kinds of site information are held in the same tables that basic assets use. For example, the `AssetPublication` table specifies which content management sites the asset type is enabled for.

When you develop the templates that display the flex assets that represent your content, you code elements that extract and display information from the `_Mungo` tables and the `MungoBlobs` table.

### Default Columns in the Flex Asset Type Database Table

As do basic asset types, each of the flex asset types has a primary storage table that takes its name from the asset type. For example, the primary table for the GE sample site asset type named product is called `Products`. The primary table for the product attribute asset type is called `PAttributes`.

Unlike the primary table for a basic asset type, the primary table for a flex asset type has only the default columns. This is because flex asset types that have attribute values do not store those values in the primary table—attribute values are stored in the `_Mungo` table for the asset type.

In general, the **default column types** in the primary table for a **flex asset type** are the **same** as the default columns in the primary storage table for a **basic asset type**. For the general list of default column types, see "Default Columns in the Basic Asset Type Database Table" on page 191.

However, there are, of course, exceptions and additions, as described in the following table:

| Column | Description |
|---|---|
| `category` | Category is **not** used in the flex asset model so there is no `category` column in any of the primary tables for flex asset types.<br>Flex assets have no need for the category feature because queries for flex assets are based on the values of their flex attributes. |
| `template` | Only the table for the flex asset member in a flex family—product, article (flex), and image (flex), for example—holds values in this column. This is because only the flex asset member in the family can have a Template asset assigned to it and be displayed on your online site. |
| `renderid` | Holds the object ID of the Template asset assigned to a flex asset. |
| `attributetype` | An additional column in the primary table for flex attribute types. It holds the name of the attribute editor that formats the input style of the attribute when it is displayed in the "New" and "Edit" forms (if there is one). |
| `flextemplateid` | An additional column in the primary table for a flex asset type (the flex asset member of a flex family.) It holds the ID of the flex definition that the flex asset was created with. |
| `flexgrouptemplateid` | An additional column in the primary table for flex parent asset types. It holds the object ID of the parent definition that the flex parent asset was created with. |

## The _Mungo Tables

The flex asset and flex parent asset types have an *`AssetType`*`_Mungo` table, where *`AssetType`* is the name of the flex asset type (and matches the name of the main storage table). Its purpose is to store the attribute values assigned to an asset when an asset of this type is created. For example the GE sample site table `Products_Mungo` holds the attribute values for product assets.

Each attribute value has a separate row.

Each row in `_Mungo` table has a value in each of the following columns:

| Column | Description |
|---|---|
| `id` | A unique ID for each attribute value, automatically generated by Content Server when the flex asset is saved and the row is created.<br>This is the table's primary key. |
| `ownerid` | The ID of the flex asset that the attribute value belongs to. (From the flex asset table: `Product`, for example.) |
| `attrid` | The ID of the attribute. (From the attribute table: `PAttributes`, for example.) |

| Column | Description |
|---|---|
| assetgroupid | If the attribute value is inherited, the ID of the parent who passed on the value. (From the parent table: ProductGroups, for example.) |

Each row in a _Mungo table also has all of the following columns but it will have a value (data) in only one of them, depending on the data type of the attribute:

| Column | Description |
|---|---|
| floatvalue | If the attribute's data type is float, the value of the attribute. |
| moneyvalue | If the attribute's data type is money, the value of the attribute. |
| textvalue | If the attribute's data type is textvalue, the value of the attribute. |
| datevalue | If the attribute's data type is date, the value of the attribute. |
| intvalue | If the attribute's data type is int, the value of the attribute. |
| blobvalue | If the attribute's data type is blob, the ID of the row in the MungoBlobs table that holds the value of the attribute. |
| urlvalue | If the attribute's data type is url, the path or url entered for the attribute. |
| assetvalue | If the attribute's data type is asset, the ID of the asset. |
| stringvalue | If the attribute's data type is float, the value of the attribute. |

Because the _Mungo tables have URL columns (see "Indirect Data Storage with the Content Server URL Field" on page 219), a default storage directory (defdir) must be set for it. You use the cc.urlattrpath property in the gator.ini file to set the defdir for your _Mungo tables.

## The MungoBlobs Table

There is one MungoBlobs table. It holds all the values for all flex attributes of type blob, for all the flex attribute types in your system. Each attribute value has a separate row in the table.

## The _AMap Tables

Flex asset and flex parent asset types have an *AssetType*_AMap table. Its purpose is to map the asset to the attributes it inherits from its parents. Then when you create a template that displays the asset on a page in your online site, you can query for assets based on any of their attributes and display any of those attributes, whether they were inherited or were directly assigned.

The _AMap table has one row for each flex asset that has a value for the inherited attribute. (However, if an attribute has more than one value, the _Mungo table has a row for each value.)

An _AMap table has the following columns:

| Column | Description |
|---|---|
| `id` | A unique ID for each row, automatically generated by Content Server when the flex asset is saved and the row is created.<br>This is the table's primary key. |
| `inherited` | The ID of the parent the attribute was inherited from, if it was inherited. (From the parent table: ProductGroups, for example.) |
| `attributeid` | The ID of the attribute. (From the attribute table: PAttributes, for example) |
| `ownerid` | The ID of the flex asset that the attribute value belongs to. (From the flex asset table: Product, for example.) |

# Assetsets and Searchstates

CS-Direct Advantage provides the `ASSETSET` and `SEARCHSTATE` method families for identifying the individual flex assets that you want to display on your online pages.

## Assetset

An **assetset** is a group of flex assets or flex parent assets **only**— not flex attributes or flex definitions or flex parent definitions. For example, in the GE sample site, you can create assetsets that contain products, articles (advanced), or images (advanced). When you code your site pages, you code statements that create assetsets and then display the assets in them.

## Searchstate

You identify which flex assets should be in an assetset by using the `SEARCHSTATE` method family in the templates for your flex assets. A **searchstate** is a set of search constraints that are applied to a list or set of flex assets. A constraint can be either a filter (restriction) based on the value of an attribute or based on another searchstate (called a nested searchstate).

A searchstate can search either the `_Mungo` tables in the database or the attribute indexes created by a search engine. This means that you can mix database and rich-text (full-text through an index) searches in the same query.

Because these tags search only the `_Mungo` table or attribute indexes for that flex asset type, using them to extract your flex assets is much more efficient than using the `ASSET` tags or the query asset.

## Assetsets and Attribute Asset Types

Content Server cannot perform searches across attribute asset types. Because assetsets are created on the basis of attribute values, only assets that share the same attribute asset type can be included in the same assetset. This is an important point to consider when you design your flex families: if you create flex asset types that do not share a common attribute asset type, you have separated your data and ensured that assets from different types cannot be included in a common assetset. And displaying assetsets is the mechanism for displaying flex assets on your delivery system.

For example, you can have two types of flex assets in the same flex family. As long as they use the same type of attributes, you can create assetsets that include assets of both types. Keep in mind, though, that a search across two types of flex assets creates a join between their `_Mungo` tables, which can deprecate performance.

In the GE sample site there are two flex asset types: article (advanced) and image (advanced). They share the same attribute asset type ("Content Attributes") and the same definitions (content definition and content parent definition). However, it is the shared attribute asset type that enables them to be included in the same assetset—even though they are two different flex asset types.

However, because articles and images do not share an attribute asset type with the GE product asset type, you cannot create an assetset that includes products and articles.

# Search Engines and the Two Asset Models

Because the data structure of the two asset models is so different, there is a key difference in the way the asset models interact with a search engine:

- A basic asset type is defined by an asset descriptor file and its primary storage table includes all of its properties as columns. To specify which fields of a basic asset type should be indexed, you must customize certain elements for the asset type. (See Chapter 15, "Designing Basic Asset Types.")
- Because "fields" for flex assets are flex attributes, which are assets, you decide which "fields" are indexed for rich-text search, attribute by attribute. Additionally, the CS-Direct Advantage application enables you to specify which attributes should be indexed with the **Search Engine** field on the attribute's "New" and "Edit" forms. You do not need to customize any elements to enable this feature.

# Tags and the Two Asset Models

The ultimate goal of creating and managing assets is to move them to your delivery system, where the code in your elements can extract them from the database and display them to your site visitors. The Content Server applications offer various "toolsets," custom tag sets, in both XML and JSP.

The toolset you use to extract assets from the database in your templates depends on the kind of asset that you are working with.

- For assets with the basic asset model, you use the `ASSET` method family.
- For the flex asset member in a flex family, you use the `ASSETSET` and `SEARCHSTATE` method families. Note that you **should not** use the **ASSET.LOAD** tag for the **flex asset member** in a flex family (product, article, and image, for example). Using `ASSET.LOAD` tag for flex assets is extremely inefficient because it retrieves all of the information for that asset from all of its tables. The `SEARCHSTATE` methods queries only the `_Mungo` table for the asset type of the flex asset and the `MungoBlobs` table.
- For recommendation assets, you use the `COMMERCECONTEXT` method family.

There are many more method families available with these products as well as an extensive set of custom tags from Content Server itself and several APIs.

For information about all the tags, see the *Content Server Tag Reference.*

See Section 4, "Site Development," for information about coding elements and site pages.

# Summary: Basic and Flex Asset Models

This section summarizes the similarities and differences between the two asset models.

## Where the Asset Models Intersect

Even though there are many differences in the way that the basic and flex asset models function, there are several points of intersection.

- No matter which asset model you are using, basic (CS-Direct) or flex (CS-Direct Advantage), you use the template and page asset types that are delivered with the CS-Direct application.
- All asset types have a status code, which means that all assets—whether they are flex or basic—can be searched for with queries based on status.
- All asset types, whether they are flex or basic, have the following configuration or administrative traits in common:
  - They must be enabled by site.
  - They must have Start Menu items configured for them before anyone can create individual instances of those types.
  - Individual instances of them can be imported with the XMLPost utility.

## Where the Asset Models Differ

The following table summarizes the major differences between the asset models:

| | Basic Asset Model | Flex Asset Model |
|---|---|---|
| **Number of database tables** | One | Several |
| **Adding fields to an asset type** | Requires a schema change. | Does not require a schema change. |
| **Links to other assets** | Through named associations and unnamed relationships. | Through flex family relationships. |
| **Subtypes** | Usually available through the Subtype item on the **Admin** tab. For more information on how subtypes are set, see "Subtype" on page 188. | Through flex definitions and flex parent definitions. |
| **Search engine indexing** | Must customize certain elements for the asset type. | Use the **Search Engine** field in the flex attribute form. |
| **Main tag families** | `ASSET`, `SITEPLAN`, and `RENDER` | `ASSETSET`, `SEARCHSTATE`, and `RENDER` |

| | Basic Asset Model | Flex Asset Model |
|---|---|---|
| **Publishing methods** | Export to Disk<br><br>Mirror to Server | Export to Server is possible, but is atypical for the flex model.<br><br>Mirror to Server |

# Summary: Asset Types

The following table lists all the asset types delivered by the Content Server modules, products, and sample sites:

| Name of asset type | Product or Sample Site |
|---|---|
| page | CS-Direct |
| template | CS-Direct |
| collection | CS-Direct |
| query | CS-Direct |
| CSElement | CS-Direct |
| SiteEntry | CS-Direct |
| link | CS-Direct |
| article | CS-Direct, Burlington Financial |
| imagefile | CS-Direct, Burlington Financial |
| stylesheet | CS-Direct, Burlington Financial |
| linkset | CS-Direct, Burlington Financial (for backward compatibility only) |
| image | CS-Direct, Burlington Financial (for backward compatibility only) |
| HelloArticle | CS-Direct, HelloAssetWorld |
| HelloImage | CS-Direct, HelloAssetWorld |
| attribute editor | CS-Direct Advantage |
| product | CS-Direct Advantage, GE Lighting |
| product attribute | CS-Direct Advantage, GE Lighting |
| product definition | CS-Direct Advantage, GE Lighting |
| product parent | CS-Direct Advantage, GE Lighting |
| product parent definition | CS-Direct Advantage, GE Lighting |
| article (flex) | CS-Direct Advantage, GE Lighting |

| Name of asset type | Product or Sample Site |
|---|---|
| image (flex) | CS-Direct Advantage, GE Lighting |
| content attribute | CS-Direct Advantage, GE Lighting |
| content definition | CS-Direct Advantage, GE Lighting |
| content parent | CS-Direct Advantage, GE Lighting |
| content parent definition | CS-Direct Advantage, GE Lighting |
| visitor attribute | Engage |
| history attribute | Engage |
| history definition | Engage |
| segment | Engage |
| recommendation | Engage |
| promotion | Engage |
| DrillHierarchy | Engage, Burlington Financial Extension |
| PDF | Engage, Burlington Financial Extension |

Chapter 12

# The Content Server Database

Just about everything in Content Server, its modules, and CS products is represented as a row in a database table.

This chapter describes the various kinds of tables and columns in the Content Server database and presents procedures for creating tables. The Content Server modules (CS-Direct, for example) and products (Engage, for example) deliver most of the tables that you need. However, if you are using Content Server to develop your own application or you need to use a table that does not hold assets—a lookup table, for example—you create that table using one of the methods described in this chapter.

This chapter contains the following sections:


- Types of Database Tables
- Types of Columns (Fields)
- Creating Database Tables
- How Information Is Added to the System Tables
- Property Files and Databases


For information about managing the data in non-asset tables, see Chapter 13, “Managing Data in Non-Asset Tables.”

**Note**

Content Server database tables used to be called “catalogs” and there are still remnants of that terminology throughout the application in table names, servlet names (CatalogManager), and the Java interfaces that you use to work with data in the database.

# Types of Database Tables

There are five types of tables in the Content Server database:

- Object tables, which hold data as objects and provides a unique identifier, automatically, for each row in the table
- Tree tables, which hold the hierarchical information about relationships between objects in object tables
- Content tables, which hold flat data and do not provide a unique identifier for each row
- Foreign tables, which can be either of the following:
  - Tables that are outside of the Content Server database but that Content Server has access to.
  - Tables that are in the Content Server database but that Content Server did not create.
- System tables, which are core Content Server application tables whose schema cannot be modified

Content Server can cache the resultsets from queries against any table in the Content Server database, including foreign tables.

## Object Tables

Object tables store data as an object and can be represented in hierarchies. Those objects can be loaded, saved, and managed with the CatalogManager API. The asset type tables for CS-Direct and CS-Direct Advantage are object tables.

The primary key for object tables is always the ID (`id`) column and that cannot be changed. When you instruct Content Server to add an object table, it creates an ID column in that table. ID is a unique identifier that Content Server assigns by default to each row as it is added to the table. For example, when someone creates a new asset with CS-Direct, Content Server determines the ID and assigns that value as the ID for that asset.

You cannot change the ID that Content Server assigns to objects (such as assets).

**Note**

When AssetMaker or Flex Family maker creates an object table for a new asset type, it creates several additional columns by default. For information about the default columns in basic asset tables, see "Default Columns in the Basic Asset Type Database Table" on page 191.

Anytime you need to store data and you want to ensure that each row of that data is uniquely identified, use an object table because Content Server handles ID generation for you.

Examples of object tables (catalogs)

- All tables that hold assets
- Many of the CS-Direct publishing tables
- The Engage tables that hold visitor data

## Tree Tables

Tree tables store information about the hierarchical relationships between object tables. In other words, object tables can be represented in hierarchies, but the hierarchy itself is stored in a tree table—the hierarchy **is** the tree.

For example, CS-Direct adds the following tree tables to the Content Server database:

- `AssetRelationTree`, which stores information about associations between assets. These associations create parent-child relationships. (For information about asset associations, see "The Flex Asset Model" on page 195.)
- `SitePlanTree`, which stores information about parent-child relationships between page assets and the assets that are referred to from those assets. This information is presented graphically on the **Site Plan** tab that is present in the Content Server interface when CS-Direct is installed.

Each row in a tree table is a node in that tree. Each node in a tree table points to two places:

- To an object in an object table, that is, to the object that it represents
- To its parent node in that tree table, unless it is a top-level node and has no parent

In other words, the object itself is stored in an object table. That object's relationships to other objects in the database (as described by the tree) are stored in the tree table as a node on a tree.

Note that children nodes point to parent nodes but parents do not point to children.

When you create a tree table, it has the following columns by default. You cannot add to or modify these columns:

| Column | Description |
|---|---|
| `nid` | The ID of the node. This is the primary key. |
| `nparentid` | The ID of the node's parent node. |
| `nrank` | A number that ranks peer or sibling nodes. For example, the `AssetRelationTree` table uses this column to determine the order of the assets that are in collections. |
| `otype` | The object type of the node. For example, in the `SitePlanTree` table (a CS-Direct table), `otype` is either the asset type "page" or the name of a site ("publication"). In the `AssetRelationTree` table (another CS-Direct table), `otype` is an asset type and is the name of the object table for assets of that type. |
| `oid` | The ID of the object that the node refers to. |
| `oversion` | Reserved for future use by FatWire. |
| `ncode` | Holds a string that has meaning in the context of what the table is being used for. For example, in the `SitePlanTree`, `ncode` is set to "placed" or "unplaced" based on whether the page asset that the node refers to has been placed or not. In the `AssetRelationTree`, `ncode` holds the name of a named association. |

## Content Tables

Content tables store data as flat data (rather than as objects) and that information cannot be organized in a hierarchy. You use content tables for simple lookup tables. For example, these are only a few of the content tables that CS-Direct adds to the Content Server database:

- `Source`, which holds strings that are used to identify the source of an article or image asset
- `Category`, which holds codes that are used to organize assets in several ways
- `StatusCode`, which holds the codes that represent the status of an asset

All three of these tables are lookup tables that the CS-Direct product uses to look up values for various columns in the asset type tables (object tables).

In another example, CS-Direct also adds a content table called `MimeType`. This table holds mimetype codes that are displayed in the **Mimetype** fields of the Burlington Financial sample site asset types named stylesheet and imagefile. The **Mimetype** fields for these asset types query the `MimeType` table for mimetype codes based on the `keyword` column in that table.

### Setting the Primary Key for a Content Table

When you create a content table, an ID column is not created for you and the primary key is not required to be ID. This is another major difference between content tables and object tables.

The `cc.contentkey` property in the `futuretense.ini` file specifies the name of the default primary key for all content tables. When you create a new content table, you are responsible for defining a column with the name specified by the `cc.contentkey property`.

However, you can override the identity of the primary key for a specific content table by adding and setting a custom property in the `futuretense.ini` file. This property must use the following format:

`cc.`*`tablename`*`Key`

For example, if you create a content table named `Books` and you want to override the default primary key so that it uses the `ISBN` column instead, you would add a property named `cc.BooksKey` and set it to `ISBN`.

## Foreign Tables

A foreign table is one that Content Server does not completely manage. For example, perhaps your site pages perform queries against a table that is populated by an ERP system and Content Server displays that information to your site visitors.

Content Server can query foreign tables and cache the resultsets just as it does for its own object and content tables. However, you must first identify that foreign table to Content Server by adding a row for it in the `SystemInfo` table. This is the **only** time you should ever modify information in the `SystemInfo` table.

Additionally, you must be sure to flush the Content Server resultset cache with a CatalogManager `flushcatalog` tag whenever the external system updates the tables that you query. Otherwise, the resultsets cached against those tables might not be up-to-date.

For information about resultset caching, see Chapter 14, “Resultset Caching and Queries.”

## System Tables

System tables are core, Content Server tables whose schema is fixed. They are implemented in Content Server by their own classes and they do not follow the rules (for caching and so on) that the other tables follow.

You can add rows to some of the system tables (either using the Content Server Management Tools forms, found on the **Admin** tab of the Content Server UI, or the Content Server Explorer tool), but you cannot add or modify the columns in these tables in any way. You also cannot add system tables to the database.

The following table lists and defines the Content Server system tables:

| Table | Description |
|---|---|
| `ElementCatalog` | Lists all the XML or JSP elements used in your system. An element is a named piece of code. For more information about the `ElementCatalog` table, see Chapter 23, "Creating Template, CSElement, and SiteEntry Assets." |
| `SiteCatalog` | Lists a page reference for each page or pagelet served by Content Server. For more information about the `SiteCatalog` table, see Chapter 23, "Creating Template, CSElement, and SiteEntry Assets." |
| `SystemACL` | Has a row for each of the access control lists (ACLs) that were created for your Content Server system. ACLs are sets of permissions to database tables.<br>For information about creating ACLs, see the *Content Server Administrator's Guide*.<br>For information about using ACLs to implement user management for your online site, Chapter 31, "User Management on the Delivery System." |
| `SystemEvents` | Has a row for each event being managed by Content Server. An event represents an action that takes place on a certain schedule.<br>Content Server inserts a row in this table when you set an event by using either the `APPEVENT` or `EMAILEVENT` tags. |
| `SystemInfo` | Lists all the tables that are in the Content Server database and any foreign tables that Content Server needs to reference. |
| `SystemItemCache` | Holds information about specific items on pages that are cached (assets, for example): the identity of the item, the page it is associated with, and the time it was cached. |
| `SystemPageCache` | Holds information about pages that are cached: the folder that it is cached to, the query used to generate the file name, the time it was cached, and the time that it should expire. |
| `SystemSeedAccess` | Registers Java classes that are external to Content Server but that Content Server has access to (includes access control). |

| Table | Description |
|---|---|
| `SystemSQL` | Holds SQL queries that you can reuse in as many pages or pagelets as necessary. You can store SQL queries in this table and then use the `ics.CallSQL` method, `CALLSQL` XML tag, the `ics:callsql` JSP tag to invoke them. Then, if you need to modify the SQL statement, you only have to modify it once. |
| `SystemUserAttr` | Stores attribute information about the users such as their e-mail addresses. Note that if you are using LDAP, this table is not used. |
| `SystemUsers` | Lists all the users who are allowed access to pages, functions, and tables. Note that if you are using LDAP, this table is not used. |

## Identifying a Table's Type

To determine the table type of any table in the Content Server database, examine the `SystemInfo` table, the system table that lists all the tables in the database.

### To determine a table type

1. Open the Content Server Explorer and log in to the Content Server database.
2. Double-click on the **SystemInfo** table.
3. In the list of tables, examine the **systable** column. The value in this column identifies the type of table represented in the row:

| Value in systable column | Definition |
|---|---|
| `yes` | system table |
| `no` | content table |
| `obj` | object table |
| `tree` | tree table |
| `fgn` | foreign table |

**Note**

If you do not have the appropriate ACLs assigned to your user name, you cannot open and examine the `SystemInfo` table.

# Types of Columns (Fields)

When you create new tables for the Content Server database, whether they hold assets or not, you can specify three general categories of field (column) types for the columns in those tables:

- Generic field types
- Database-specific field types
- The Content Server URL field

## Generic Field Types

Generic field types refers to field types that work in any DBMS that Content Server supports. They are mapped to be compliant with JDBC standards. Therefore, if your Content Server system changes to a different DBMS, your database is still valid.

When you use generic, JDBC-compliant field types, you can use the CatalogManager API (`CATALOGMANAGER` XML or JSP tags, or the `ics.CatalogManager` Java method) to modify and maintain the data in your tables.

The following table contains a complete list of the Content Server generic field types and the database properties (from the `futuretense.ini` file) that define their data types. Refer to this list whenever you create a new table with the Content Server Management Tools forms, found on the **Admin** tree in the Content Server user interface, or the CatalogManager API:

| Field Type | Description | Property |
|---|---|---|
| `CHAR(n)` | A short string of exactly *n* characters. | `cc.char` |
| `VARCHAR(n)` | A short string of up to *n* characters. For example, `VARCHAR(32)` means that this column can hold a string of up to 32 characters. | `cc.varchar` and `cc.maxvarcharsize` (The maximum value that you can set for `cc.varchar` depends on the value of the `cc.maxvarcharsize` property.) |
| `DATETIME` | A date/time combination. | `cc.datetime` |
| `TEXT` | A `LONGVARCHAR`, a variable-length string of up to 2,147,483,647 | `cc.bigtext` |
| `IMAGE` | One binary large object (blob). | `cc.blob` |
| `SMALLINT` | A 16-bit integer, that is, an integer from -32,768 to +32,767. | `cc.smallint` |
| `INTEGER` | A 32-bit integer, that is, an integer from -2,147,483,648 to +2,147,483,647. | `cc.integer` |

| Field Type | Description | Property |
|---|---|---|
| BIGINT | A 64-bit integer, that is, integers having up to 19 digits. | cc.bigint |
| NUMERIC(*L*,*P*) | A floating-point (real) number, having a total number of *L* significant digits of which up to *P* significant digits are fractional. For example, NUMERIC(5,2) could represent a number such as 806.35 but could not accurately represent a number such as 25693.2283 | cc.numeric |
| DOUBLE | A double precision type. | cc.double |

In addition to defining the column type, you must specify which of the following column constraints applies to the column:

| Constraint | Description |
|---|---|
| NULL | It can hold a null value, that is, it can be left empty. |
| NOT NULL | It cannot hold a null value, that is, it cannot be left empty |
| UNIQUE NOT NULL | It must hold a value that is guaranteed to be unique in this table. |
| PRIMARY KEY NOT NULL | Marks the primary key column in a content table. You cannot set this column constraint for an object table. |

When you use AssetMaker to create an object table for a new asset type (CS-Direct) or when you create new flex attributes (CS-Direct Advantage), the data types for those items are different than the ones listed here.

For more information about the data types for columns for basic asset types, see "Storage Types for the Columns" on page 271. For information about the data types for flex attributes, see "Data Types for Attributes" on page 199.

## Database-Specific Field Types

You can use database-specific field (column) types in your tables. However, if you use field types that are specific to one kind of DBMS (that is, types that have not been mapped to a JDBC standard), note the following:

- You may not be able to use the CatalogManager API on those tables.
- If you ever change your DBMS you must also modify your tables.

For a complete list of field types specific to the DBMS that you are using, consult your DBMS documentation.

## Indirect Data Storage with the Content Server URL Field

Object and content tables in the Content Server database have a unique characteristic: columns can store their data indirectly, which means that you can store large bits of data externally to the DBMS but within the data repository.

To create such a column, you must use a column name that begins with the letters `url`. When you use the letters `url` as the first three letters of a column name, Content Server treats that column as an indirect data column.

Why use a URL field? For the following reasons:

- When the DBMS you are using does not support fields that are large enough to accommodate the size of the data that you want to store there
- If the DBMS you are using does not support enough fields in an individual table to contain the data that you want to store
- Because performance is degraded when fields of a large size are selected

**Note**

If the size of the data you wish to store in a URL column exceeds the value set for the `cc.maxvarcharsize` property in the `futuretense.ini` file, then that data is stored in the database as `bigtext`, instead of being stored indirectly (as a file referenced by a pointer in the database).

## The Default Storage Directory (defdir)

Any table with a URL column must have a default storage directory specified for it. This directory is where the values entered into the column are actually stored.

The phrase "default storage directory" is shortened to the word **defdir** in several places in the product. For example, the `defdir` column in the `SystemInfo` table holds the name of the default storage directory for tables with URL columns; one of the forms for the AssetMaker utility presents a **defdir** field; and so on.

The value entered into a URL field is actually a relative path to a file. Why a relative path? Because the value in a URL field is appended to the value of the table's defdir setting.

The way that you set the defdir value for the tables that you create depends on the applications you have and what you are doing:

- If you create a new Content Server table with the Content Server Management Tools forms, found on the **Admin** tree in the Content Server user interface, and your table has a URL field, you enter the value for defdir in the **File Storage Directory** field in the "Add Catalog" (table) form.
- If you create a new Content Server table with the CatalogManager API, you use the `uploadDir` argument to set the value of defdir.
- If you create a new basic asset type, you specify the value of the defdir in the **defdir** field on the AssetMaker form. (Note that all tables that hold basic assets have a URL column and must have a defdir value set.)
- If you create a new flex asset type, you do **not** specify the value of the defdir for the URL column in the flex asset's `_Mungo` table. This value is obtained from a property that was set when your CS-Direct Advantage application was installed. **Never** change the value of that property.

**Caution**

After a table with a URL column is created, do not attempt to change or modify the `defdir` setting for the table in any way. If you do, you will break the link between the storage directory and the URL column, which means that your data can no longer be retrieved.

For information about creating URL fields, see the following procedures and examples:

- "Creating Database Tables" on page 220
- The upload field examples for basic asset types, starting with "Upload Example 1: A Standard Upload Field" on page 284
- The procedure for creating flex attributes of type blob in the section "Creating Flex Attributes of Type Blob (Upload Field)" on page 319.

# Creating Database Tables

This section describes how to create object, tree, and content tables and how to register foreign tables (that is, identify them to Content Server). You cannot create or modify system tables.

## Creating Object Tables

There are three ways to create object tables:

- Create tables that hold basic asset types. You must use AssetMaker, a CS-Direct utility located on the **Admin** tab. AssetMaker creates the object table for the asset type as well as the CS-Direct forms that you use to create assets of that type. For more information, see Chapter 15, "Designing Basic Asset Types."
- Create tables that hold flex asset types. You must use Flex Family Maker, a CS-Direct Advantage utility located on the **Admin** tab. For more information, see Chapter 16, "Designing Flex Asset Types."
- Create an object table that does **not** hold assets. Use the Content Server Management Tools, found on the **Admin** tree in the Content Server user interface, (or Content Server Explorer).

**To create an object table that does *not* hold assets**

1. Open your browser and enter this address:

   `http://your_server/Xcelerate/LoginPage.html`

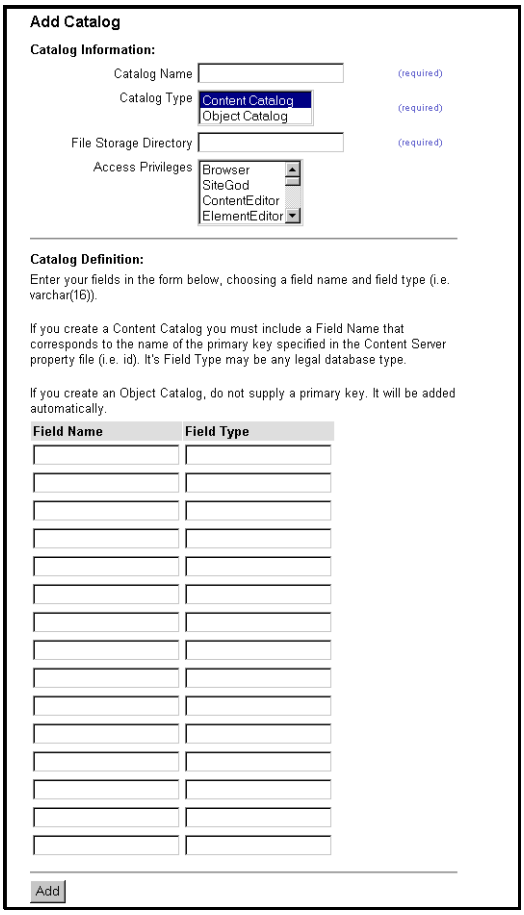
2. Enter your login name and password and click **Login**.

   The Content Server interface appears.

3. Select the **Admin** tab and then select **Content Server Management Tools > Content Catalog** (table).

4. In the “ContentCatalogManagement” form, select **Add Catalog** (table) and click **OK**.

   The “Add Catalog” (table) form appears.




5. Click in the **Catalog Name** (table name) field and enter a name. Do not use the name of a table that already exists. You can enter up to 64 alphanumeric characters, including the underscore (_) character but not including spaces.
6. Click in the **Catalog Type** (table type) field and select **Object Catalog**.
7. If your table will have a URL column (an upload column), click in the **File Storage Directory** field (that is, the defdir) and enter the path to the file directory that will store the data from the URL column. If the directory does not exist yet, Content Server will create it for you.
8. Click in the **Access Privileges** field to select which ACLs (access control lists) a user must have in order to access this table. For information about ACLs, see the *Content Server Administrator’s Guide*.
9. Click in the **Field Name** column and enter the name of the field. Remember that to create a URL column that stores data as a file located in an external directory, you must start the field name with the letters `url`. If you are creating a URL column, be sure that you have specified the file storage directory (defdir) for the data stored in this field (see step 7 of this procedure).

10. Click in the **Field Type** column and specify both the data type and column constraint for the column. Include a space between the data type and the column constraint.

    For example: `VARCHAR(32) NULL` or `INTEGER NOT NULL`.

    For a list of the valid data types and the column constraints, see "Generic Field Types" on page 217.

11. Repeat steps 9 and 10 for each column in your new table.

12. Click the **Add** button.

    Content Server adds the table to the database.

To verify that your table has been added, open Content Server Explorer and examine the `SystemInfo` table. Your new table should be included in the list with its `systable` column set to `obj`. If you specified a file storage directory, it is listed in the `defdir` column.

## Managing Data in Object Tables

There are several ways to modify and manage the data in object tables.

To create and modify **assets**, you use CS-Direct, CS-Direct Advantage, and Engage applications. To extract assets from the database and then display them to the visitors of your delivery system, you use the CS-Direct, CS-Direct Advantage, and Engage XML and JSP tags.

You can enter data into object tables that do not hold assets in one of the following ways:

- Programmatically, by coding forms with the `ics.CatalogManager` Java method or the `CATALOGMANAGER` XML and JSP tags, the `OBJECT` XML and JSP tags, and the Content Server SQL methods and tags, that prompt users for information and then to write that information to the database
- Manually by using either the Content Server Explorer tool or a form in the Content Server Management Tools forms to add rows to the table.

The following chapter, Chapter 13, "Managing Data in Non-Asset Tables," presents information about the CatalogManager API and examples of adding rows to tables that do not hold assets.

# Creating Tree Tables

If you are using CS-Direct or any of the other CS modules or products, it is unlikely that you would need to create a tree table.

Tree tables are managed by the TreeManager servlet. To create a tree table (catalog), you use the `ICS.TreeManager` Java method or the `TREEMANAGER` XML or JSP tags. You cannot create a tree table (catalog) with the Content Server Management Tools.

For example:

```
<TREEMANAGER>
   <ARGUMENT NAME="ftcmd" VALUE="createtree"/>
   <ARGUMENT NAME="treename" VALUE="ExampleTree"/>
</TREEMANAGER>
```

For a list of the columns that are created for tree tables, see "Tree Tables" on page 213. For information about the TreeManager methods and tags, see the *Content Server Tag Reference*.

## Managing Data in Tree Tables

The CS-Direct application and the other Content Server modules and products manage all the data in their tree tables. You should never attempt to manually modify information in any of the CS-Direct tree tables.

If you have any tree tables that you created to manage relationships between your own object tables (that is, object tables that do not store assets), you use the `ICS.TreeManager` Java method or the `TREEMANAGER` XML or JSP tags. These tags and methods use an `FTValList` parameter, which describes the tree operation to be performed.

The following chapter, Chapter 13, "Managing Data in Non-Asset Tables," presents information about the CatalogManager API and examples of adding rows to tables that do not hold assets.

# Creating Content Tables

**To create a content table, use the Content Server Management Tools**

1. If you are using any of the Content Server applications, open your browser and enter this address:

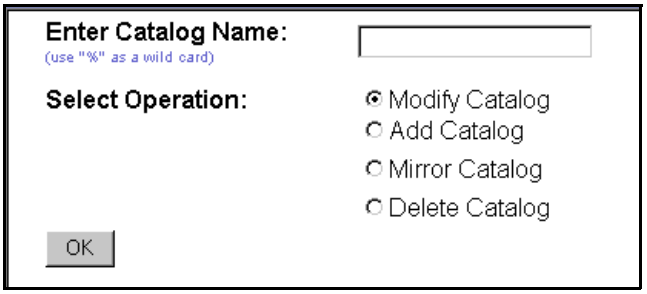
   `http://`*`your_server`*`/Xcelerate/LoginPage.html`

   Note that *`your_server`* is the name of the web server that you are using with Content Server.

2. Enter your login name and password and click **Login**.

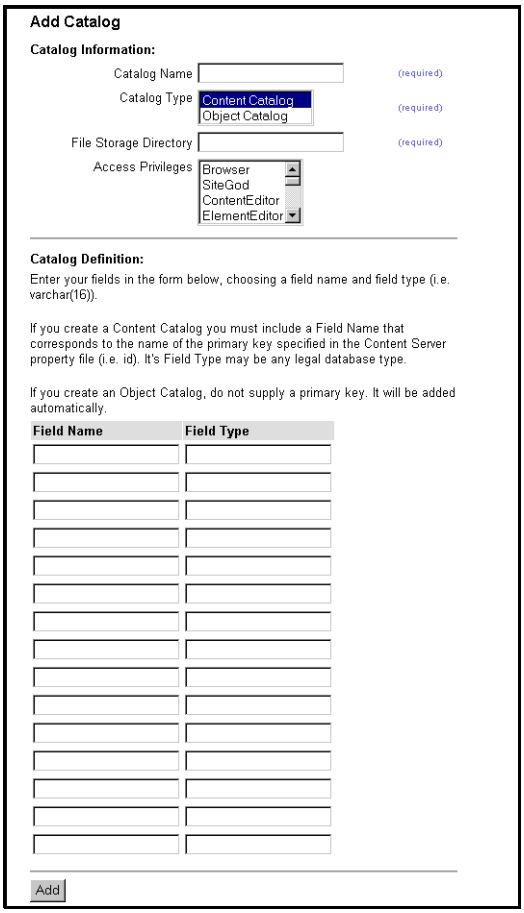
   The Content Server user interface appears.

3. Select the **Admin** tab and then select **Content Server Management Tools > Content Catalog (table)**.



4. In the **ContentCatalogManagement** window, select **Add Catalog** (table) and click **OK**.

   The **Add Catalog** (table) window appears.




5. Click in the **Catalog Name** (table name) field and enter a name. Do not use the name of a table that already exists. You can enter up to 64 alphanumeric characters, including the underscore (_) character but not including spaces.
6. Click in the **Catalog Type** (table type) field and select **Content Catalog**.
7. If your table will have a URL column, click in the **File Storage Directory** field and enter the path to the file directory that will store the data from the URL column. If the directory does not exist yet, Content Server will create it for you.
8. Click in the **Access Privileges** field to select which ACLs (access control lists) a user must have in order to access this table. For information about ACLs, see the *Content Server Administrator's Guide*.
9. Click in the **Field Name** field and enter the name of the field. Remember that to create a URL column that stores data as a file located in an external directory, you must start the field name with the letters `url`. If you are creating a URL column, be sure that you have specified the file storage directory (defdir) for the data stored in this field (see step 7 of this procedure).

10. Click in the **Field Type** column and specify both the data type and column constraint for the column. Include a space between the data type and the column constraint.

    For example: `VARCHAR(32) NULL` or `INTEGER NOT NULL`.

    Remember that you must specify a primary key column and that it must exactly match either the setting for the `cc.contentkey` property or a custom property specified for this table in the `futuretense.ini` file.

    For example: `INTEGER PRIMARY KEY NOT NULL`

11. Repeat steps 9 and 10 for each column in your new table.

12. Click the **Add** button.

    Content Server adds the table to the database.

To verify that your table has been added, open the Content Server Explorer tool and examine the `SystemInfo` table. Your new table should be included in the list with the value in its `systable` column set to `no`. If you specified a file storage directory, it is listed in the `defdir` column.

When you add non-asset tables to facilitate some function of your site, you then need to either customize your asset forms in CS-Direct or create your own forms to enter and manipulate that data.

## Managing Data in Content Tables

There are several ways to modify and manage the data in content tables.

Most of the CS-Direct content tables have a CS-Direct form available from the Admin tab that you can use to edit or add data. For example, `Source` and `Category`. (`MimeType`, however, does not.)

You can enter data into your custom content tables in one of the following ways:

- If the table was created such that users or visitors supply data that is written into the table, you code forms with the `ics.CatalogManager` Java method or the `CATALOGMANAGER` XML and JSP tags along with the Content Server SQL methods and tags—to prompt users for information and to then write it to the database programmatically.
- If the table is a simple lookup table that facilitates some function on your site, you enter data into it manually by using either the Content Server Explorer utility or the Content Server Management Tools window to add rows to the table.

Chapter 13, “Managing Data in Non-Asset Tables,” presents information about the CatalogManager API and examples of adding rows to tables that do not hold assets.

# Registering a Foreign Table

Registering a foreign table means identifying the table to Content Server by adding a row for the table to the `SystemInfo` table. Note that this is the only condition in which you should ever add a row to the `SystemInfo` table or change information held in the `SystemInfo` table in any way.

### To register a foreign table

1. Open Content Server Explorer and log in to the Content Server database.
2. Double-click on the **SystemInfo** table.

3. Right-click the header for the **tblname** column and then select **New** from the pop-up menu.

   A new row appears.
4. In the new row, click in the **tblname** column and enter the name of the table.
5. Click in the **defdir** column and enter the path to the table.
6. Click in the **systable** column and enter `fgn`.
7. Click in the **acl** column and enter the names of the ACLs that have access to the table.
8. Select **File > Save All.**

### Managing Data in a Foreign Table

You can use the `ics.CatalogManager` Java method or the `CATALOGMANAGER` XML and JSP tags and the Content Server SQL methods and tags to interact with a foreign table. When you use these methods or tags to update data in the foreign table, Content Server can flush its resultset cache as needed.

If you use a method external to Content Server to update a foreign table, you must be sure to also use the `CATALOGMANGER` command `flushcatalog` to instruct Content Server to flush the resultset cache for that table.

## How Information Is Added to the System Tables

You cannot create system tables and with very few exceptions, you should always use the Content Server Management Tools to add rows to the system tables that you are allowed to add rows to. The way that information is added to each system table varies, as described in the following table:

| Table | Method of Adding Information |
|---|---|
| `SiteCatalog` | There are several ways that page entries are added to this table:<br>• When you create a Template asset, CS-Direct automatically creates a page entry for it in the `SiteCatalog` table.<br>• When you create a SiteEntry asset, CS-Direct automatically creates a page entry for it in the `SiteCatalog` table.<br>• You can use the Content Server Explorer tool or the “Site” form in the Content Server Management Tools interface.<br>Note that if you want to set or modify page cache settings for page entries, it is easier to use forms in the Content Server interface than it is to use Content Server Explorer.<br>See Section 4, “Site Development,” for information about designing pages. |

| Table | Method of Adding Information |
|---|---|
| `ElementCatalog` | There are several ways that elements are added to this table:<br>• When you create a Template asset, CS-Direct automatically creates an entry for it in the `ElementCatalog` table.<br>• When you create a CSElement asset, CS-Direct automatically creates an entry for it in the `ElementCatalog` table.<br>• You can use the Content Server Explorer tool to add non-asset elements.<br>For information about coding elements and pages, see Chapter 27, "Coding Elements for Templates and CSElements." |
| `SystemACL` | The "ACL" form in the **Content Server Management Tools** node.<br>For more information, see the *Content Server Administrator's Guide*. |
| `SystemEvents` | Content Server adds a row to this table for each event that is designated when an `APPEVENT` tag, `EMAILEVENT` tag, or Java API equivalent is invoked from an element. |
| `SystemInfo` | Do **not** add or modify information to this table.<br>The **only exception** to this rule is if you need to identify a foreign table to Content Server. |
| `SystemSQL` | The Content Server Explorer tool.<br>For information about the various kinds of queries that are available, see Chapter 14, "Resultset Caching and Queries." |
| `SystemUsers` | The "User" form in the **Content Server Management Tools** node. |
| `SystemUserAttr` | The "User" form in the **Content Server Management Tools** node. |

# Property Files and Databases

There are several properties in the `futuretense.ini` file that control the connection to the Content Server database. These properties specify the configuration of the database and establish a privileged and non-privileged user connection between the database and the application server.

All of the database properties were configured for your system when your system was installed. By default, all commands operate on the Content Server database identified in the `futuretense.ini` file. The location of this file depends on which application server your system is using:

- With the Sun ONE Application Server, `futuretense.ini` is located in the `/ias/APPS` directory.
- With any other application server, `futuretense.ini` is located in the `FutureTense` directory.

Chapter 13

# Managing Data in Non-Asset Tables

This chapter describes how to interact with Content Server database tables that do not hold assets.

There are two ways to work with the data in your custom, non-asset tables:

- Programmatically, using the tags and methods for the CatalogManager API to code forms for data entry and management
- Manually, by using the Content Server Explorer tool or the "Content" form in the Content Server Management Tools to manually add rows and data to those rows.

This chapter contains the following sections:


- Methods and Tags
- Coding Data Entry Forms
- Managing the Data Manually
- Deleting Non-Asset Tables


To work with assets, you must log in to the Content Server interface and use the asset forms provided by the CS-Direct, CS-Direct Advantage, and Engage applications.

To add large numbers of assets programmatically, use the XMLPost utility, as described in Chapter 20, "Importing Assets of Any Type" and Chapter 21, "Importing Flex Assets."

# Methods and Tags

This section provides an overview of the tags and methods that you use to program how you manage data in non-asset tables and how you interact with those tables in general.

## Writing and Retrieving Data

CatalogManager is the Content Server servlet that manages content and object tables in the database and the TreeManager servlet manages tree tables in the database.

- To access the CatalogManager servlet, you can use the `ics.CatalogManager` Java method, the `CATALOGMANAGER` XML tag, or the `ics:catalogmanager` JSP tag.
- To access the TreeManager servlet, you can use the `ics.TreeManager` Java method, the `TREEMANAGER` XML tag, or the `ics:treemanager` JSP tag.

These methods and tags take name/value pairs from arguments that specify the operation to perform and the table to perform that operation on.

### CatalogManager

The `ics.CatalogManager` java method, the `CATALOGMANAGER` XML tag, and the `ics:catalogmanager` JSP tag support a number of attributes that operate on object and content tables. The key attribute is `ftcmd`. By setting `ftcmd` to `addrow`, for example, you tell CatalogManager to add one row to the catalog.

CatalogManager security, when enabled, prevents users with the `DefaultReader` ACL from accessing CatalogManager. You enable CatalogManager security by setting the `secure.CatalogManager` property, found in the `futuretense.ini` file, to `true`. Note that your session will be dropped if you attempt to log out of CatalogManager when CatalogManager security is enabled.

These are the main `CATALOGMANAGER` XML tag's attributes, passed as argument name/value pairs, that modify the contents of a row or a particular field in a row:

| argument name="ftcmd" value= | Description |
|---|---|
| `addrow` | Adds a single row to a table. |
| addrows | Adds more than one row to a table. |
| deleterow | Deletes a row from a table. You must specify the primary key column for the row. |
| deleterows | Deletes more than one row from a table. You must specify the primary key for the rows. |
| `replacerow` | Deletes the existing row in a table and replaces the row with the specified information. |
| `replacerows` | Replaces multiple rows in a table. If a value is not specified for a column, the column value is cleared |
| `updaterow` | Performs a query against a given table and displays records from a table. The rows displayed match the criteria specified by the value of the parameters. |

| argument name="ftcmd" value= | Description |
|---|---|
| updaterow2 | Like updaterow, updates values in columns for a row in a table; however, where you cannot clear columns with updaterow, updaterow2 allows you to clear columns if there is no value for the specified column (for example, if there is no related field in the form). |
| updaterows | Modifies field values for multiple rows in a table. |
| updaterows2 | Like updaterows, modifies field values for multiple rows in a table; however, where you cannot clear columns with updaterows, updaterows2 allows you to clear columns if there is no value for the specified column (for example, if there is no related field in the form). |

For more information and a complete list of the CatalogManager commands, see the *Content Server Tag Reference*. For information about the ics.CatalogManager Java method, see the *Content Server Javadoc*.

## Tree Manager

Here are the main ics.TreeManager commands. Note that these operations manipulate data in the tree table only—but do not affect the objects that the tree table nodes refer to.

| Name | Description |
|---|---|
| addchild | Given a parent node, add a child node. |
| addchildren | Add multiple child nodes. |
| copychild | Copy a node and its children to a different parent. All copied nodes point to the same objects. |
| createtree | Create a tree table. |
| delchild | Delete a node and its child nodes. |
| delchildren | Delete multiple nodes. |
| deletetree | Delete a tree table. |
| findnode | Find a node in a tree. |
| getchildren | Get all child nodes. |
| getnode | Get node and optionally object attributes. |
| getparent | Get the nodes parent. |
| listtrees | Get the list of all tree tables. |
| movechild | Move node and its child nodes to a different parent. |
| nodepath | Return parent; child path to a node. |
| setobject | Associate a different object with the node. |

| Name | Description |
|---|---|
| validatenode | Verify that a node is in a tree. |
| verifypath | Verify that a given path exists in a tree. |

For information about the `ics.TreeManager` method, see the *Content Server Javadoc.*

For information about the XML and JSP `TREEMANAGER` tags, see the *Content Server Tag Reference*.

## Querying for Data

There are three methods, with XML and JSP tag counterparts, to help your code query for and select content:

| Method | XML tag | JSP tag | Description |
|---|---|---|---|
| ics.SelectTo | SELECTTO | ics:selectto | Performs a simple select against a single table. |
| ics.SQL | EXECSQL | ics:sql | Executes an inline SQL statement (embedded in the code). |
| ics.CallSQL | CALLSQL | ics:callsql | Executes a SQL statement that is stored as a row in the SystemSQL table. |

To use `ics.CallSQL` (or the tags), you code SQL statements and then paste them into the `SystemSQL` table.

By storing the actual queries in the `SystemSQL` table and calling them from the individual pages (like you call a pagename or an element), you keep them out of your code, which makes it easier to maintain the SQL used by your site. If you want to change the SQL, you do not have to fix it in every place that you use it —you can just edit it in the `SystemSQL` table and every element that calls it now calls the edited version.

The `ics.CallSQL` and `ics.SQL` methods can execute any legal SQL commands. If a SQL statement does not return a usable list, Content Server will generate an error. If you choose to use SQL to update or insert data, you must include code that explicitly flushes the resultsets cached against the appropriate tables using the `ics.FlushCatalog` method.

## Lists and Listing Data

A number of `ICS` methods create lists. The `SelectTo` method, for example, returns the results of a simple SQL query in a list whose columns reflect the items in the WHAT clause and whose rows reflect matches against the table.

The `IList` interface can be used to access a list from Java. The lists are available by name using XML or JSP, and values can be iterated using the `LOOP` tag.

The lists created by Content Server point to underlying resultsets created from querying the database. Although the lists do not persist across requests, the resultsets do because if are cached.

**Note**

Be sure to configure resultset caching appropriately. If the resultset of a query is cached, the list points to a copy of the resultset. If the resultset is not cached, the list points directly at the resultset which can cause database connection resource difficulties.

You can create your own list for use in XML or JSP by implementing a class based on the IList interface. Then your application or page can transform data prior to returning an item in a list or to create a single list from many lists.

The following methods manage lists:

| Method | Description |
|---|---|
| `ics.GetList` | Returns an `IList`, given the name of the list. |
| `ics.CopyList` | Copies a list. |
| `ics.RenameList` | Renames an existing list. |
| `ics.RegisterList` | Registers a list by name with Content Server so that you can reference the list from an XML or JSP element or by using the `GetList` method. |

For an example implementation of an `IList`, see `SampleIList.java` in the `Samples` folder on your Content Server system.

# Coding Data Entry Forms

This section provides code samples that illustrate how to code forms that accept information entered by a user or visitor and to then write that information to the database using the Content Server methods and tags.

The examples in this section describe adding a new row, deleting a row, and querying for and then editing an existing row. Each example shows a version for XML, JSP, and Java.

## Adding a Row

A simple algorithm for adding a row is as follows:

1. Display a form requesting information for each of the fields in a row.
2. Write that form data to the table.

The following example adds a row to a fictitious table named `EmployeeInfo`. This table has the following columns:

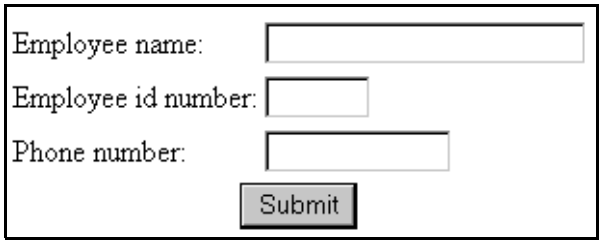
| Field | Data type |
|---|---|
| `id` | `VARCHAR(6)` |
| `phone` | `VARCHAR(16)` |
| `name` | `VARCHAR(32)` |

This example presents code from the following elements:

- `addrowFORM`, an XML element that displays a form that requests an employee ID number, phone number, and name.
- `addrowXML`, `addrowJSP`, and `addrowJAVA`, three versions of an element that writes the information entered by the employee to the `EmployeeInfo` table

### The addrowFORM Element

The `addrowFORM` element displays a form that asks the user to enter information. It looks like this:




This is the code that creates the form:

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Documentation/CatalogManager/addrowFORM
-->
```

```
<form ACTION="ContentServer" method="post"
   REPLACEALL="CS.Property.ft.cgipath">
   <input type="hidden" name="pagename" value="Documentation/
   CatalogManager/addrow"/>

<table>
<tr>
   <td>Employee name:</td>
         <td><input type="text" value="" name="EmployeeName"
   size="22" maxlength="32"/></td>
</tr>
<tr>
   <td>Employee id number:</td>
   <td><input type="text" value="" name="EmployeeID" size="6"
   maxlength="6"/></td>
</tr>
<tr>
   <td>Phone number:</td>
   <td><input type="text" value="" name="EmployeePhone" size="12"
   maxlength="16"/></td>
</tr>
<tr>
   <td colspan="2"><input type="submit" name="submit"
   value="Submit"/></td>
</tr>
</table>

</form>
</FTCS>
```

Notice that the maxlength modifiers in `<INPUT>` limit the length of each input to the maximum length that was defined in the schema.

The user fills in the form and clicks the **Submit** button. The information gathered in the form and the pagename of the `addrow` page (see the first `input type` statement in the preceding code sample) is sent to the browser. The browser sends the pagename to Content Server. Content Server looks it up in the `SiteCatalog` table and then invokes that page entry's root element.

## Root Element for the addrow Page

The root element of the `addrow` page is responsible for adding the information passed from the `addrowFORM` element to the database — that is, for adding a row to the `EmployeeInfo` table and populating that row with the information passed from the `addrowFORM` element.

There can only be one root element for a Content Server page (that is, an entry in the `SiteCatalog` table). This section shows three versions of the root element for the `addrow` page:

- `addrowXML.xml`
- `addrowJSP.jsp`
- `addrowJAVA.jsp`

### addrowXML

This is the code in the XML version of the root element:

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Documentation/CatalogManager/addrowXML
-->

<SETVAR NAME="errno" VALUE="0"/>
<CATALOGMANAGER>
   <ARGUMENT NAME="ftcmd" VALUE="addrow"/>
   <ARGUMENT NAME="tablename" VALUE="EmployeeInfo"/>
   <ARGUMENT NAME="id" VALUE="Variables.EmployeeID"/>
   <ARGUMENT NAME="phone" VALUE="Variables.EmployeePhone"/>
   <ARGUMENT NAME="name" VALUE="Variables.EmployeeName"/>
</CATALOGMANAGER>
errno=<CSVAR NAME="Variables.errno"/><br/>
</FTCS>
```

**Note**

The example code can use the `CATALOGMANAGER` tag because the fictitious table, `EmployeeInfo`, has Content Server generic field types. `addrowXML` might not work if `EmployeeInfo` has database-specific field types. For more information, see "Generic Field Types" on page 217

### addrowJSP

This is the code in the JSP version of the root element:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%@ taglib prefix="ics" uri="futuretense_cs/ics.tld" %>
<%//
// Documentation/CatalogManager/addrowJSP
//%>
<%@ page import="COM.FutureTense.Interfaces.*" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<cs:ftcs>

<ics:setvar name="errno" value="0"/>
<ics:catalogmanager>
         <ics:argument name="ftcmd" value="addrow"/>
         <ics:argument name="tablename" value="EmployeeInfo"/>
         <ics:argument name="id"
            value='<%=ics.GetVar("EmployeeID")%>'/>
         <ics:argument name="phone"
            value='<%=ics.GetVar("EmployeePhone")%>'/>
         <ics:argument name="name"
   value='<%=ics.GetVar("EmployeeName")%>'/>
</ics:catalogmanager>
```

```
errno=<ics:getvar name="errno"/><br/>

</cs:ftcs>
```

### addrowJAVA

This is the code in the Java version of the root element:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%//
// Documentation/CatalogManager/addrowJAVA
//%>
<%@ page import="COM.FutureTense.Interfaces.*" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<cs:ftcs>

<!-- user code here -->
<%
ics.SetVar("errno","0");
FTValList vl = new FTValList();
vl.put("ftcmd","addrow");
vl.put("tablename","EmployeeInfo");
vl.put("id",ics.GetVar("EmployeeID"));
vl.put("phone",ics.GetVar("EmployeePhone"));
vl.put("name",ics.GetVar("EmployeeName"));
ics.CatalogManager(vl);
%>
errno=<%=ics.GetVar("errno")%><br />

</cs:ftcs>
```

## Deleting a Row

The following example deletes a row from the fictitious `EmployeeInfo` table described in the "Adding a Row" on page 234 is section.

This section presents code from the following elements:

- `deleterowFORM`, an XML element that displays a form that requests an employee name to delete from the `EmployeeInfo` table
- `deleterowXML`, `deleterowJSP`, and `deleterowJAVA`, elements that delete a row from the `EmployeeInfo` table based on the information sent to it from the `deleterowFORM` element

### The deleterowFORM Element

The `deleterowFORM` element displays a form that asks the user to enter an employee name. This is the code that creates the form:

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Documentation/CatalogManager/deleterowFORM
-->
```

```
<form ACTION="ContentServer" method="post"
REPLACEALL="CS.Property.ft.cgipath">
<input type="hidden" name="pagename" value="Documentation/
CatalogManager/deleterow"/>

<table>
<tr>
         <td>Employee name:</td>
         <td><input type="text" value="Barton Fooman"
   name="EmployeeName" size="22" maxlength="32"/></td>
</tr>
<tr>
         <td colspan="2"><input type="submit" name="submit"
   value="submit"/></td>
</tr>
</table>
</form>
</FTCS>
```

The user enters an employee name and clicks the **Submit** button. The employee name and the pagename for the `deleterow` page (see the first `input type` statement in the preceding code sample) are sent to the browser.

The browser sends the pagename to Content Server. Content Server looks it up in the `SiteCatalog` table and then invokes that page entry's root element.

## Root Element for the deleterow Page

The root element of the `deleterow` page is responsible for deleting a row from the `EmployeeInfo` table, based on the employee name that is sent to it from the `deleterowFORM` element.

There can only be one root element for a Content Server page (that is, an entry in the `SiteCatalog` table). This section shows three versions of the root element for the `deleterow` page:

- `deleterowXML.xml`
- `deleterowJSP.jsp`
- `deleterowJAVA.jsp`

### deleterowXML

This is the code in the XML version of the element:

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Documentation/CatalogManager/deleterowXML
-->
<SETVAR NAME="errno" VALUE="0"/>

<CATALOGMANAGER>
   <ARGUMENT NAME="ftcmd" VALUE="deleterow"/>
   <ARGUMENT NAME="tablename" VALUE="EmployeeInfo"/>
   <ARGUMENT NAME="tablekey" VALUE="name"/>
   <ARGUMENT NAME="tablekeyvalue" VALUE="Variables.EmployeeName"/>
```

```
</CATALOGMANAGER>

errno=<CSVAR NAME="Variables.errno"/><br/>
</FTCS>
```

### deleterowJSP

This is the code in the JSP version of the element:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%@ taglib prefix="ics" uri="futuretense_cs/ics.tld" %>
<%//
// Documentation/CatalogManager/deleterowJSP
//%>
<%@ page import="COM.FutureTense.Interfaces.*" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<cs:ftcs>

<!-- user code here -->
<!-- user code here -->
<ics:setvar name="errno" value="0"/>
<ics:catalogmanager>
         <ics:argument name="ftcmd" value="deleterow"/>
         <ics:argument name="tablename" value="EmployeeInfo"/>
         <ics:argument name="name"
   value='<%=ics.GetVar("EmployeeName")%>'/>
</ics:catalogmanager>

errno=<ics:getvar name="errno"/><br />

</cs:ftcs>
```

### deleterowJAVA

This is the code in the Java version of the element:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%//
// Documentation/CatalogManager/deleterowJAVA
//%>
<%@ page import="COM.FutureTense.Interfaces.*" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<cs:ftcs>

<%
ics.SetVar("errno","0");
FTValList vl = new FTValList();
vl.put("ftcmd","deleterow");
vl.put("tablename","EmployeeInfo");
vl.put("name",ics.GetVar("EmployeeName"));
ics.CatalogManager(vl);
%>
errno=<%=ics.GetVar("errno")%><br />

</cs:ftcs>
```

## Querying a Table

The following sample elements query the fictitious `EmployeeInfo` table for an employee's name, extract the employee name and displays it in a browser, prompts the user to edit the information, and then writes the edited information to the database.

This section presents code from the following elements:

- `SelectNameForm`, an XML element that displays a form that requests an employee's name.
- Three versions of the `QueryEditRowForm` element (XML, JSP, and Java), an element that locates the employee name and loads the information about that employee into a form that the employee can use to edit his or her information
- Three versions of the `QueryEditRow` element (XML, JSP, and Java), an element that writes the newly edited information to the database.

### The SelectNameForm Element

The `SelectNameForm` element displays a simple form that requests the name of the employee who is altering his employee information. This is the code:

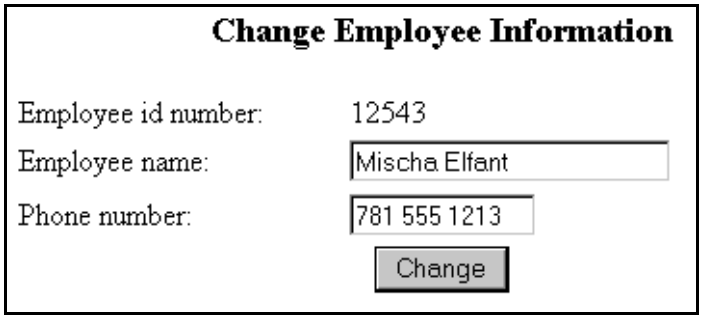
```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Documentation/CatalogManager/SelectNameForm
-->

<form ACTION="ContentServer" method="post">
<input type="hidden" name="pagename" value="Documentation/
CatalogManager/QueryEditRowForm"/>
<TABLE>
<TR>
   <TD>Employee name: </TD>
   <TD><INPUT type="text" value="" name="EmployeeName" size="22"
   maxlength="32"/></TD>
</TR>
<TR>
   <TD COLSPAN="100%" ALIGN="CENTER">
   <input type="submit" name="doit" value="Submit"/></TD>
</TR>
</TABLE>
</form>
</FTCS>
```

When the employee clicks the **Submit** button, the information gathered in the **Employee Name** field and the name of the `QueryEditRowForm` page (see the first `input type` statement in the preceding code sample) is sent to the browser.

The browser sends the pagename to Content Server. Content Server looks up the pagename in the `SiteCatalog` table, and then invokes that page entry's root element, `QueryEditRowForm`.

## The Root Element for the QueryEditRowForm Page

The root element for the `QueryEditRowForm` page locates the row in the `EmployeeInfo` table that matches the string entered in the **Employee Name** field and then loads the data from that row into a new form. The employee can edit her name and phone number but cannot edit her id. The form looks like this:




There can only be one root element for a Content Server page (that is, an entry in the `SiteCatalog` table). This section shows three versions of the root element for the `QueryEditRowForm` page:

- `QueryEditRowFormXML.xml`
- `QueryEditRowFormJSP.jsp`
- `QueryEditRowFormJAVA.jsp`

### QueryEditRowFormXML

This is the code in the XML version of the element:

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Documentation/CatalogManager/QueryEditRowFormXML
-->

<SETVAR NAME="errno" VALUE="0"/>
<SETVAR NAME="name" VALUE="Variables.EmployeeName"/>
<SELECTTO FROM="EmployeeInfo"
      WHERE="name"
      WHAT="*"
      LIST="MatchingEmployees"/>

<IF COND="Variables.errno=0">
<THEN>
   <form ACTION="ContentServer" method="post">
   <input type="hidden" name="pagename" value="Documentation/
   CatalogManager/QueryEditRow"/>
   <input type="hidden" name="MatchingID"
   value="MatchingEmployees.id" REPLACEALL="MatchingEmployees.id"/
   >
   <TABLE>
   <TR>
         <TD COLSPAN="100%" ALIGN="CENTER">
         <H3>Change Employee Information</H3>
         </TD>
```

```
   </TR>
   <TR>
         <TD>Employee id number: </TD>
         <TD><CSVAR NAME="MatchingEmployees.id"/></TD>
   </TR>
         <TR>
         <TD>Employee name: </TD>
         <TD><INPUT type="text" value="MatchingEmployees.name"
   name="NewEmployeeName" size="22" maxlength="32"
   REPLACEALL="MatchingEmployees.name"/></TD>
   </TR>
   <TR>
         <TD>Phone number: </TD>
         <TD><INPUT type="text" value="MatchingEmployees.phone"
   name="NewEmployeePhone" size="12" maxlength="16"
   REPLACEALL="MatchingEmployees.phone"/></TD>
   </TR>
   <TR>
         <TD colspan="100%" align="center">
         <input type="submit" name="doit" value="Change"/></TD>
   </TR>
   </TABLE>
   </form>
</THEN>
<ELSE>
   <P>Could not find this employee.</P>
   <CALLELEMENT NAME="Documentation/CatalogManager/
   SelectNameFormXML"/>
</ELSE>
</IF>
</FTCS>
```

When the employee clicks the **Change** button, the information gathered from the two fields and the name of the `QueryEditRow` page is sent to the browser.

The browser sends the pagename and the field information to Content Server. Content Server looks up the pagename in the `SiteCatalog` table, and then invokes that page entry's root element.

### QueryEditRowFormJSP

This is the code in the JSP version of the element:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%@ taglib prefix="ics" uri="futuretense_cs/ics.tld" %>
<%//
// Documentation/CatalogManager/QueryEditRowFormJSP
//%>
<%@ page import="COM.FutureTense.Interfaces.*" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<cs:ftcs>

<ics:setvar name="errno" value="0"/>
<ics:setvar name="name" value='<%=ics.GetVar("EmployeeName")%>'/>
```

```
<ics:selectto table="EmployeeInfo"
      where="name"
      what="*"
      listname="MatchingEmployees"/>

<ics:if condition='<%=ics.GetVar("errno").equals("0")%>'>
<ics:then>
   <form action="ContentServer" method="post">
   <input type="hidden" name="pagename" value="Documentation/
   CatalogManager/QueryEditRow"/>
   <input type="hidden" name="MatchingID" value="<ics:listget
   listname='MatchingEmployees' fieldname='id'/>"/>
         <TABLE>
         <TR>
            <TD COLSPAN="100%" ALIGN="CENTER">
               <H3>Change Employee Information</H3>
            </TD>
         </TR>
         <TR>
            <TD>Employee id number: </TD>
            <TD><ics:listget listname='MatchingEmployees
            fieldname='id'/></TD>
         </TR>
         <TR>
            <TD>Employee name: </TD>
            <TD><INPUT type="text" value="<ics:listget
               listname='MatchingEmployees' fieldname='name'/>"
               name="NewEmployeeName" size="22" maxlength="32"/></TD>
         </TR>
         <TR>
            <TD>Phone number: </TD>
            <TD><INPUT type="text" value="<ics:listget
                  listname='MatchingEmployees' fieldname='phone'/>"
                  name="NewEmployeePhone" size="12"
   maxlength="16"/>
            </TD>
         </TR>
         <TR>
            <TD colspan="100%" align="center">
            <input type="submit" name="doit" value="Change"/></TD>
         </TR>
         </TABLE>
         </form>
</ics:then>
<ics:else>
         <P>Could not find this employee.</P>
         <ics:callelement element="Documentation/CatalogManager/
            SelectNameForm"/>
</ics:else>
</ics:if>

</cs:ftcs>
```

When the employee clicks the **Change** button, the information gathered from the two fields and the name of the `QueryEditRow` page is sent to the browser.

The browser sends the pagename and the field information to Content Server. Content Server looks up the pagename in the `SiteCatalog` table, and then invokes that page entry's root element.

### QueryEditRowFormJAVA

This is the code in the Java version of the element:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%//
// Documentation/CatalogManager/QueryEditRowFormJAVA
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%//
// Documentation/CatalogManager/QueryEditRowFormJAVA
//%>
<%@ page import="COM.FutureTense.Interfaces.*" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<cs:ftcs>

<!-- user code here -->
<%
ics.SetVar("errno","0");
ics.SetVar("name",ics.GetVar("EmployeeName"));
StringBuffer errstr = new StringBuffer();
IList matchingEmployees = ics.SelectTo("EmployeeInfo",// tablename
                                       *", // what
                                       "name", // where
                                       "name", // orderby
                                       1, // limit
                                       null, // ics list name
                                       true, // cache?
                                       errstr); // error StringBuffer

if ("0".equals(ics.GetVar("errno")) && matchingEmployees!=null &&
matchingEmployees.hasData())
{
         %>
         <form action="ContentServer" method="post">
         <input type="hidden" name="pagename"
   value="Documentation/CatalogManager/QueryEditRow"/>
         <%
         String id = matchingEmployees.getValue("id");
         String name = matchingEmployees.getValue("name");
         String phone = matchingEmployees.getValue("phone");
         %>
         <input type="hidden" name="MatchingID" value="<%=id%>"/>
         <TABLE>
         <TR>
            <TD COLSPAN="100%" ALIGN="CENTER">
               <H3>Change Employee Information</H3>
```

```
            </TD>
         </TR>
         <TR>
            <TD>Employee id number: </TD>
            <TD><%=id%></TD>
         </TR>
         <TR>
            <TD>Employee name: </TD>
            <TD><INPUT type="text" value="<%=name%>"
   name="NewEmployeeName" size="22" maxlength="32"/></TD>
         </TR>
         <TR>
            <TD>Phone number: </TD>
            <TD><INPUT type="text" value="<%=phone%>"
   name="NewEmployeePhone" size="12" maxlength="16"/></TD>
         </TR>
         <TR>
            <TD colspan="100%" align="center">
            <input type="submit" name="doit" value="Change"/></TD>
         </TR>
         </TABLE>
         </form>
<%
}
else
{
   %>    <P>Could not find this employee.</P>
<%
         ics.CallElement("Documentation/CatalogManager/
   SelectNameForm",null);
}
%>
</cs:ftcs>
```

When the employee clicks the **Change** button, the information gathered from the two fields and the name of the `QueryEditRow` page is sent to the browser.

The browser sends the pagename and the field information to Content Server. Content Server looks up the pagename in the `SiteCatalog` table, and then invokes that page entry's root element.

## The Root Element for the QueryEditRow Page

The root element for the `QueryEditRow` page writes the information that the employee entered into the **Employee Name** and **Phone number** fields and updates the row in the database.

There can only be one root element for a Content Server page (that is, an entry in the `SiteCatalog` table). This section shows three versions of the root element for the `QueryEditRow` page:

- `QueryEditRowXML.xml`
- `QueryEditRowJSP.jsp`
- `QueryEditRowJAVA.jsp`

### QueryEditRowXML

This is the code in the XML version of the element:

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Documentation/CatalogManager/QueryEditRowXML
-->

<SETVAR NAME="errno" VALUE="0"/>

<CATALOGMANAGER>
   <ARGUMENT NAME="ftcmd" VALUE="updaterow"/>
   <ARGUMENT NAME="tablename" VALUE="EmployeeInfo"/>
   <ARGUMENT NAME="id" VALUE="Variables.MatchingID"/>
   <ARGUMENT NAME="name" VALUE="Variables.NewEmployeeName"/>
   <ARGUMENT NAME="phone" VALUE="Variables.NewEmployeePhone"/>
</CATALOGMANAGER>

<IF COND="Variables.errno=0">
<THEN>
   <P>Successfully updated the database.</P>
</THEN>
<ELSE>
   <P>Failed to update the information in the database.</P>
</ELSE>
</IF>
</FTCS>
```

### QueryEditRowJSP

This is the code in the JSP version of the element:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%@ taglib prefix="ics" uri="futuretense_cs/ics.tld" %>
<%//
// Documentation/CatalogManager/QueryEditRowJSP
//%>
<%@ page import="COM.FutureTense.Interfaces.*" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<cs:ftcs>

<ics:setvar name="errno" value="0"/>

<ics:catalogmanager>
         <ics:argument name="ftcmd" value="updaterow"/>
         <ics:argument name="tablename" value="EmployeeInfo"/>
         <ics:argument name="id"
   value="<%=ics.GetVar("MatchingID")%>"/>
         <ics:argument name="name"
         value='<%=ics.GetVar("NewEmployeeName")%>'/>
         <ics:argument name="phone"
         value='<%=ics.GetVar("NewEmployeePhone")%>'/>
```

```
</ics:catalogmanager>

<ics:if condition='<%=ics.GetVar("errno").equals("0")%>'>
<ics:then>
   <P>Successfully updated the database.</P>
</ics:then>
<ics:else>
   <p>failed to update the information in the database.
   errno=<ics:getvar name='errno'/></p>
</ics:else>
</ics:if>

</cs:ftcs>
```

### QueryEditRowJAVA

This is the code in the Java version of the element:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%//
// Documentation/CatalogManager/QueryEditRowJAVA
//%>
<%@ page import="COM.FutureTense.Interfaces.*" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<cs:ftcs>

<!-- user code here -->
<%
ics.SetVar("errno","0");

FTValList args = new FTValList();
args.put("ftcmd","updaterow");
args.put("tablename","EmployeeInfo");
args.put("id",ics.GetVar("MatchingID"));
args.put("name",ics.GetVar("NewEmployeeName"));
args.put("phone",ics.GetVar("NewEmployeePhone"));

ics.CatalogManager(args);

if("0".equals(ics.GetVar("errno")))
{
   %><P>Successfully updated the database.</P><%
}
else
{
   %><p>failed to update the information in the database.
   errno=<ics:getvar name='errno'/></p><%
}
%>
</cs:ftcs>
```

## Querying a Table with an Embedded SQL Statement

The following example shows another method of searching for a name in a table. This example also searches the fictitious `EmployeeInfo` table, returning the rows that match the string supplied by a user, but this time the code uses a SQL query rather than a `SELECTTO` statement.

This section presents code from the following elements:

- `QueryInlineSQLForm`, an XML element that displays a form that requests a movie title
- Three versions of the `QueryInlineSQL` element (XML, JSP, and Java), an element that searches the `EmployeeInfo` table for names that contain the string entered by the user in the preceding form

### QueryInlineSQLForm

The `QueryInlineSQL` element displays a simple form that requests the name to use to search the `EmployeeInfo` table for. This is the code:

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Documentation/CatalogManager/QueryInlineSQLForm
-->
<form ACTION="ContentServer" method="post">
<input type="hidden" name="pagename" value="Documentation/
CatalogManager/QueryInlineSQL"/>

<table>
<tr>
         <td>Employee Name:</td>
         <td><input type="text" value="Foo,Bar"
   name="EmployeeName" size="22" maxlength="32"/></td>
</tr>
<tr>
         <td colspan="2"><input type="submit" name="submit"
   value="submit"/></td>
</tr>
</table>

</form>
</FTCS>
```

When the user clicks the **Submit** button, the information gathered in the **Employee Name** field and the name of the `QueryInlineSQL` page is sent to the browser.

The browser sends the pagename of the `QueryInlineSQL` page to Content Server. Content Server looks up the pagename in the `SiteCatalog` table, and then invokes that page entry's root element.

### The Root Element for the QueryInlineSQL Page

The root element for the `QueryInlineSQL` page executes an inline SQL statement that searches the `EmployeeInfo` table for entries that match the string sent to it from the `QueryInlineSQLForm` element.

There can only be one root element for a Content Server page (that is, an entry in the `SiteCatalog` table). This section shows three versions of the root element for the `QueryInlineSQL` page:

- `QueryInlineSQLXML.xml`, which uses the `EXECSQL` XML tag to create the SQL query
- `QueryInlineSQLJSP.jsp`, which uses the `ics:sql` JSP tag to create the SQL query
- `QueryInlineSQLJAVA.jsp`, which uses the `ics.CallSQL` Java method to create the SQL query

### QueryInlineSQLXML

This is the code in the XML version of the element:

```
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Documentation/CatalogManager/QueryInlineSQLXML
-->

<SETVAR NAME="tablename" VALUE="EmployeeInfo"/>

<SQLEXP OUTSTR="MySQLExpression"
            TYPE="OR"
            VERB="LIKE"
            STR="Variables.EmployeeName"
            COLNAME="name"/>

<EXECSQL
         SQL="SELECT id,name,phone FROM Variables.tablename WHERE
   Variables.MySQLExpression"
               LIST="ReturnedList"
               LIMIT="5"/>

<table border="1" bgcolor="99ccff">
<tr>
         <th>id</th>
         <th>name</th>
         <th>phone</th>
</tr>

<LOOP LIST="ReturnedList">
         <tr>
            <td><CSVAR NAME="ReturnedList.id"/></td>
            <td><CSVAR NAME="ReturnedList.name"/></td>
            <td><CSVAR NAME="ReturnedList.phone"/></td>
         </tr>
</LOOP>
</table>

</FTCS>
```

Notice that the SQL statement is not actually embedded in the `EXECSQL` tag. Instead, a preceding `SQLEXP` tag creates a SQL expression which is passed as an argument to the

EXECSQL call. The EXECSQL tag performs the search and returns the results to the list variable named ReturnedList.

Also notice that the first line of code in the body of the element creates a variable named tablename and sets the value to EmployeeInfo, the name of the table that is being queried. This enables CatalogManager to cache the resultset against the correct table.

### QueryInlineSQLJSP

This is the code in the JSP version of the element:

```
<?xml version="1.0" ?>
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%@ taglib prefix="ics" uri="futuretense_cs/ics.tld" %>
<%//
// Documentation/CatalogManager/QueryInlineSQLJSP
//%>
<%@ page import="COM.FutureTense.Interfaces.*" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<cs:ftcs>

<!-- user code here -->
<ics:setvar name="tablename" value="EmployeeInfo"/>

<%
// no ics:sqlexp tag, must do in java
String sqlexp =
ics.SQLExp("EmployeeInfo","OR","LIKE",ics.GetVar("EmployeeName"),"
name");
String sql = "SELECT id,name,phone FROM
"+ics.GetVar("tablename")+" WHERE "+sqlexp;
%>
<ics:sqltable='<%=ics.GetVar("tablename")%>'
               sql='<%=sql%>'
               listname="ReturnedList"
               limit="5"/>

<table border="1" bgcolor="99ccff">
<tr>
         <th>id</th>
   <th>name</th>
         <th>phone</th>
</tr>

<ics:listloop listname="ReturnedList">
   <tr>
   <td><ics:listget listname="ReturnedList" fieldname="id"/></td>
   <td><ics:listget listname="ReturnedList" fieldname="name"/></
   td>
   <td><ics:listget listname="ReturnedList" fieldname="phone"/></
   td>
   </tr>
</ics:listloop>
```

```
</table>

</cs:ftcs>
```

Notice that the SQL statement is not actually embedded in the `ics:sql` tag. Instead, a preceding Java expression creates a SQL expression that is passed as an argument to the `ics:sql`call. (The code example uses Java because there is no JSP equivalent of the `SQLEXP` tag.) The `ics:sql` tag performs the search and returns the results to the list variable named `ReturnedList`.

Also notice that the first line of code in the body of the element creates a variable named `tablename` and sets the value to `EmployeeInfo`, the name of the table that is being queried. This enables CatalogManager to cache the resultset against the correct table.

### QueryInlineSQLJava

This is the code in the Java version of the element:

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%//
// Documentation/CatalogManager/QueryInlineSQLJAVA
//%>
<%@ page import="COM.FutureTense.Interfaces.*" %>
<%@ page import="COM.FutureTense.Util.ftMessage"%>
<%@ page import="COM.FutureTense.Util.ftErrors" %>
<cs:ftcs>

<%

ics.SetVar("tablename","EmployeeInfo");

String sqlexp = ics.SQLExp(ics.GetVar("tablename"),"OR","LIKE",
ics.GetVar("EmployeeName"),"name");
String sql = "SELECT id,name,phone FROM
"+ics.GetVar("tablename")+" WHERE "+sqlexp;
StringBuffer errstr = new StringBuffer();

IList list =
ics.SQL(ics.GetVar("tablename"),sql,null,5,true,errstr);

%>

<table border="1" bgcolor="99ccff">
<tr>
         <th>id</th>
         <th>name</th>
         <th>phone</th>
</tr>

<%

while (true)
{
%>
```

```
            <tr>
               <td><%=list.getValue("id")%></td>
               <td><%=list.getValue("name")%></td>
               <td><%=list.getValue("phone")%></td>
            </tr>
<%
            if (list.currentRow() == list.numRows())
               break;
            list.moveTo(list.currentRow()+1);
}
%>

</table>
</cs:ftcs>
```

Notice that the SQL statement is not actually embedded in the `ics.SQL` statement. Instead, a preceding ics.SQLExp statement creates a SQL expression which is passed as an argument to the `EXECSQL` call. The `ics.SQL` statement performs the search and returns the results to the list variable named `ReturnedList`.

Also notice that this code also creates a variable named `tablename` and sets the value to `EmployeeInfo` (the name of the table that is being queried), before the code for the query. This enables CatalogManager to cache the resultset against the correct table.

# Managing the Data Manually

You can add data to a table manually with either the Content Server Explorer tool or the forms in the Content Server Management Tools.

Content Server Explorer is the right choice in the following situations:

- If you are creating a page entry for a new page in the `SiteCatalog` table.
- If you are creating a row for an element in the `ElementCatalog` table and are coding that element with the editor in Content Server Explorer.
- If you need to add a small amount of data to a table that you have created to support some function of your site. That is, to add a small amount of data to a table that does not hold assets. (For example, to add rows to the `MimeType` table.)

Content Server Explorer has online help that you can use if you need information about adding, editing, or deleting rows. Additionally, Chapter 4, “Programming with Content Server,” in this guide describes how to add page entries to the `SiteCatalog` table and elements to the `ElementCatalog` table.

The Content Server Management Tools are the right choice in the following situations:

- To add users or ACLs to the system.
- When you want to modify the cache settings for a page entry in the `SiteCatalog` table. Typically it is easier to complete this task in the “ContentManagement” form than it is to enter the information directly into the column using Content Server Explorer.

The Content Server Management Tools are documented in the *Content Server Administrator’s Guide*.

# Deleting Non-Asset Tables

Note that if you delete a non-asset table that is being revision tracked from the database, the tracking table will not be removed. To prevent this, be sure that you disable revision tracking for the table before deleting it.

Chapter 14

# Resultset Caching and Queries

This chapter shows you how to configure resultset caching, create queries that allow CatalogManager to accurately cache resultsets, and flush resultsets from the cache.

This chapter contains the following sections:


- Overview
- Specifying the Table Name
- Flushing the Resultset Cache
- Enabling Resultset Caching over inCache
- Configuring Resultset Caching
- Summary

# Overview

Whenever the database is queried, Content Server serves a resultset—either a cached resultset or an uncached resultset. Resultset caching reduces the load on your database and improves the response time for queries.

The `futuretense.ini` file provides global properties that set the size and timeout periods for all resultsets. You can add table-specific properties to the `futuretense.ini` file that override the default settings on a table-by-table basis. These custom properties enable you to fine-tune your systems for peak performance.

## Caching Frameworks

As of version 7.6 Patch 2, Content Server provides the option to store resultsets within the inCache framework instead of hash tables. Both caching frameworks use Java memory. You can switch between frameworks by setting the `rsCacheOverinCache` property in `futuretense.ini` to either `true` or `false`.

When resultset caching over inCache is enabled, the **System Tools** node, on the **Admin** tab of the Advanced interface, displays the resultset over inCache tool, which provides statistical information about the caches and their contents. Enabling resultset caching over inCache does not require enabling either page caching or asset caching over inCache.

Instructions for enabling resultset caching over inCache are provided in this chapter, on page 260. Resultset caching in hash tables is enabled by default. Both caching models can be configured by setting properties in the `futuretense.ini` file, as shown in "Configuring Resultset Caching" on page 260. For comprehensive information about the inCache framework, its caching models, and system tools, see the *Content Server Administrator's Guide.*

## Database Queries

There are several ways to query the Content Server database for information. For example:

- With the `ics.SelectTo` Java method, `SELECTTO` XML tag, or `ics:selectto` JSP tag
- With the `selectrow` command of the `ics.CatalogManager` Java method, the `CATALOGMANAGER` XML tag, and the `ics:catalogmanager` JSP tag
- With the `ics.SQL` Java method, `EXECSQL` XML tag, or `ics:sql` JSP tag
- With the `ics.CallSQL` Java method, `CALLSQL` XML tag, or `ics:callsql` JSP tag
- Through the "Search" forms in the Content Server interface
- With a query asset
- With a `SEARCHSTATE` XML or JSP tag (flex assets only)

## How Resultset Caching Works

When the database is queried, the resultset from the query is cached if resultset caching is enabled. Then, if someone runs the same query and the data in the table remains unchanged since the last time the query was run, Content Server serves the information from the resultset cache rather than querying the database again. Serving a resultset from cache is always faster than performing another database lookup.

The resultset cache is either a hash table or the inCache framework, depending on how the `resultsetCacheOverinCache` property in `futuretense.ini` is configured. The resultsets are organized by the name of the table that was associated with the query that generated the resultset. In other words, resultsets are cached against a table name.

Each time a table is updated (from either the Content Server interface or through a CatalogManager command in your custom elements), all the resultsets in the cache for that table are flushed. Resultsets are cached in the context of a single Java VM. Although Java VMs do not share resultsets, Content Server sends a signal to all the Java VMs in a cluster to flush the resultsets when they become invalid, as long as the synchronization feature has been enabled on all servers in the cluster.

## Reducing the Load on the Database

Resultset caching reduces the load on your database in two ways:

- Serving a cached resultset does not open a database connection. Content Server attempts to obtain a resultset from the cache before it contacts the database. If the correct resultset exists, no contact is made with the database.
- When resultset caching is enabled but the appropriate resultset is not cached, Content Server obtains the resultset, stores it in the cache as an object, and then releases the database connection.

  When resultset caching is not enabled, Content Server cannot close the database connection until either the online page is completely rendered or the uncached resultset is explicitly flushed from the scope with a `flush` tag. When this occurs, your available database connections can be quickly used up (even on a relatively simple page).

As a general rule, resultset caching should be enabled for all of your database tables. Although there are times when you might need to limit either the number of resultsets that are cached or the length of time that they are cached for, it is rarely a good idea to disable resultset caching altogether.

**Note**

Never disable resultset caching on the `ElementCatalog` table. If you do, the performance of your system will suffer greatly, especially if you are using JSP in any of your elements.

# Specifying the Table Name

There must always be a table name associated with a query so that the resultset can be cached against that table. Then, whenever that table is updated through the Content Server interface or your own custom elements, CatalogManager flushes all the resultsets associated with that table.

The way that the table name is specified for a resultset depends on the type of query you are running. The following sections describe the most commonly used methods for querying the database and how you specify the table name for such a query.

## SELECTTO

When you use the ics.SelectTo Java method, SELECTTO XML tag, or ics:selectto JSP tag, you must specify the name of the table with a FROM parameter (clause). For example:

```
<SELECTTO FROM="EmployeeInfo"
          WHERE="name"
          WHAT="*"
          LIST="MatchingEmployees"/>
```

In this case, EmployeeInfo is the name of the table that is being queried and is the name of the table that the resultset is cached against. Whenever the EmployeeInfo table is updated, CatalogManager flushes all the resultsets cached against it.

## EXECSQL

EXECSQL allows you to execute an inline SQL statement. You specify the table or tables that you want to cache the resultset against using the TABLE parameter. If you specify multiple tables (by using a comma-separated list), the resultset will be cached against the first table in the list. Note that this means the resultset will be cached based on the resultset cache settings specified for the first table, including timeout and maximum size.

CatalogManager deletes outdated resultsets as the specified tables are updated.

For example, the following query caches the resultset against the article table:

```
<EXECSQL SQL="SELECT article.headline, images.imagefile FROM
article,images WHERE article.id='FTX1EE17FWB' AND
images.id='FTK9384FWW'" LIST="sqlresult" TABLE="article,images"/>
```

## CALLSQL

When you use the ics.CallSQL Java method, CALLSQL XML tag, or ics:callsql JSP tag to invoke a SQL query that is stored in the SystemSQL table, the table name is set by the query's entry (row) in the SystemSQL table.

The SystemSQL table has a deftable column that identifies the table name that the resultset from the query should be cached against. You can specify multiple tables by putting a comma-separated list of tables in the deftable column. The first table in the list is the table that the query is cached against.

Each query stored in the table must have a value in the deftable column. If it does not, CatalogManager cannot store the resultsets accurately, which means they cannot be flushed when it is necessary. Note that the table name must identify an existing table. If you enter the name of a table that does not exist yet or if you misspell the name of the table, the resultset cannot be cached correctly.

## Search Forms in the Content Server Interface

The "Search" forms that you use to look for assets in the Content Server interface search by asset type. The resultsets from the search form queries are stored against the primary storage table for assets of that type.

For example, for the Burlington Financial sample site asset named article, those resultsets are cached against the Article table; for page assets, it is the Page table; and so on.

## Query Asset

Query assets can return assets of one type only. When you create a query asset, you specify what kind of asset the query asset returns in the **Result of Query** field: articles, or imagefiles and so on.

When that query asset is used on a page in the online site, Content Server stores the resultset against the table name of the primary storage table for the asset type that the query asset returns: `Article` or `Imagefile`, and so on.

## SEARCHSTATE

The CS-Direct Advantage `SEARCHSTATE` XML and JSP tags create a set of search constraints that are applied to a list or set of flex assets (created with the `ASSETSET` tags). A constraint can be either a filter (restriction) based on the value of an attribute or based on another searchstate (called a nested searchstate).

You use the `SEARCHSTATE` and `ASSETSET` tags to extract and display flex assets or flex parent assets (not definitions or flex attributes) on your online pages for your visitors.

Content Server caches the resultsets of searchstates against the `_Mungo` table for the flex asset type. For example, if the searchstate returns the GE Lighting sample site flex asset named product, the resultset is cached against the `Products_Mungo` table.

When you configure the delivery system, be sure to add resultset caching properties for all of your `_Mungo` tables.

# Flushing the Resultset Cache

In most cases, data is written to the database through the CatalogManager API, which flushes the resultset cache when it is appropriate to do so. For example:

- If you use Content Server Explorer to add a row to a table (the `SiteCatalog` table or the `ElementCatalog` table, for example), CatalogManager flushes all the resultsets cached against that table.
- If you use a form in the Content Server interface to add or edit an asset, a source, a category, a workflow process, a user, an ACL and so on, CatalogManager flushes the resultsets cached against the tables that are written to.
- If you use CatalogManager commands in an element of your own to update a single table, Catalog Manager automatically flushes the resultsets cached against that table.
- If you use CatalogManager commands in an element of your own to update multiple (joined) tables, Catalog Manager automatically flushes the resultsets cached against the joined tables.
- If you use the `CALLSQL` tag to execute a SQL statement that is stored in the `SystemSQL` table, Catalog Manager automatically updates the resultsets cached against the table or tables specified in the `deftable` column.

# Enabling Resultset Caching over inCache

1. Ensure that the `linked-cache.xml` configuration file is in the application server's classpath (`WEB-INF/classes` directory).
2. In the `futuretense.ini` file, set the `rsCacheOverinCache` property to `true`.
3. Configure resultset caching properties as shown in the next section.

# Configuring Resultset Caching

Table 6 lists properties in the `futuretense.ini` file that control the resultset cache for all tables that are not assigned table-specific caching properties. The properties apply to resultsets that are cached within the inCache framework and in hash tables.

To change these properties, open the `futuretense.ini` file with the Property Editor utility and modify them. For information about using the Property Editor, see Chapter 8, "Content Server Tools and Utilities."

**Table 6:** Properties That Control the Resultset Cache

| property | description |
|---|---|
| `cc.cacheResults` | The default number of resultsets to cache in memory. Note that this does not mean the number of records in a resultset, but the number of resultsets.<br>**Caution:** Unless you are debugging, do **not** set this property to `0` or `-1`. If you do, the Content Server interface will fail to save assets properly. (Setting this property to `0` or `-1` disables resultset caching for all tables that do not have their own caching properties configured.) |
| `cc.cacheResultsTimeout` | Specifies the number of minutes to keep a resultset cached in memory.<br>Setting this property to `-1` means that there is no timeout value for tables that do not have their own caching properties configured. |
| `cc.cacheResultsAbs` | Specifies how expiration time for resultsets in the resultset cache is calculated.<br>• If this property is set to `true`, the expiration time for a resultset is absolute. For example, if `cc.cacheResultsTimeout` is set to 5 minutes, then 5 minutes after the resultset was cached, it is flushed from the cache.<br>• If this property is set to `false`, the expiration time for a resultset is based on its idle time. For example, if `cc.cacheResultsTimeout` is set to 5 minutes, the resultset is flushed from the cache 5 minutes after the last time it was requested rather than 5 minutes since it was originally cached. |

## Table-Specific Properties

CatalogManager not only uses the properties described in Table 6, but also checks the `futuretense.ini` file to determine if there are any custom resultset caching properties for specific tables.

You can create three resultset caching properties for each table in the Content Server database. They work the same way as do the default properties defined in Table 6. All system tables have these properties set for them.

Syntax for the custom properties is the following:

```
cc.<tablename>CSz=<number of resultsets>
cc.<tablename>Timeout=<number of minutes>
cc.<tablename>Abs=<true or false>
```

These custom properties enable you to fine-tune your systems for peak performance.

**Note**

If an asset type is enabled for revision tracking, and you wish to cache the resultsets of asset versions, use the properties above, but add `_t` after `<tablename>`:

```
cc.<tablename>_tCSz=<number of resultsets>
cc.<tablename>_tTimeout=<number of minutes>
cc.<tablename>_tAbs=<true or false>
```

For more information about resultset caching, see the *Property Files Reference*.

Open the `futuretense.ini` file in the Property Editor utility and add table-specific properties for each table that you want to control. (For information about using the Property Editor utility, see Chapter 8, “Content Server Tools and Utilities.”)

## Planning Your Resultset Caching Strategy

Before you configure resultset caching for your database, create a spreadsheet of all the tables in your Content Server database, assemble a team of developers and database administrators, and discuss what the settings should be for all of your systems—development, management, testing, and delivery. One strategy is to identify a large group of similar tables for which you can use the default properties, and then add table-specific properties for the exceptions. To tune your delivery system for the best performance possible, however, it is likely that you will create custom properties for each table in the database on that system—at the very least, 50–100 of them.

**Note**

If you set the `com.fatwire.logging.cs.cache.resultset` property, debugging messages about the resultset cache are written to the `futuretense.txt` log file. (Set the property in either the `commons-logging.properties` file, or in `log4j.properties`, depending on which logging framework you are using.)

# Summary

Resultset caching reduces the load on your database and improves the response time for queries. Be sure to do the following:

- Set the default resultset caching properties in the `futuretense.ini` file to values that make sense on each of your systems—development, management, testing, and delivery.
- Add table-specific resultset caching properties to the `futuretense.ini` file to fine-tune the performance of all of your systems—development, management, testing, and delivery.
- Provide the correct table name for all of your queries so the resultsets are cached correctly and can be flushed correctly.

Chapter 15

# Designing Basic Asset Types

As mentioned in Chapter 11, “Data Design: The Asset Models,” the data model for basic asset types is one database table per asset type. Each asset of that type is stored in that table.

You create new basic asset types with the AssetMaker utility. Typically you create them on a development system and then, when they are ready, you migrate your work from the development system to the management and delivery systems.

This chapter contains the following sections:


- The AssetMaker Utility
- Creating Basic Asset Types
- Deleting Basic Asset Types
- Images and eWebEditPro

# The AssetMaker Utility

**AssetMaker** is the utility that you use to create new basic asset types.

Your first step is to define the basic asset type *outside Content Server* by coding an `xml` file called an **asset descriptor file**, using AssetMaker XML tags. The asset descriptor file defines properties for the new asset type. The term **property** means both a column in a database table and a field in a Content Server entry form.

Your next step is to upload the file to Content Server and use AssetMaker to create two items: a database table for the new asset type, and the CS-Direct elements which generate the forms that you and others will use when working with assets of the new type (creating, editing, copying, and so on). Figure 4 on page 266 shows the relationship of a database table to a form (a content-entry form, in this example) and how columns are interpreted as fields when the form is rendered.

## How AssetMaker Works

Using AssetMaker to create a new basic asset type involves four general steps:

1. Code the asset descriptor file.

   This chapter describes asset descriptor files and coding them. The *Content Server Tag Reference* includes a chapter that describes all of the AssetMaker tags.

2. Upload the file.

   When you upload the asset descriptor file, AssetMaker creates a row in the `AssetType` table and copies the asset descriptor file to that row.

3. Create the table.

   When you click the **Create Asset Table** button, AssetMaker does the following:

   - Parses the asset descriptor file.
   - Creates the primary storage table for assets of that type. The name of the table matches the name of the asset type identified in the asset descriptor file. The data type of each column is defined by statements in the file as well.

     In addition to the columns defined in the asset descriptor file, AssetMaker creates default columns that CS-Direct needs to function correctly.
   - Adds a row for the new table to the `SystemInfo` table.

     All asset tables are object tables so the value in the `systable` column is set to `obj`.

     All asset tables have URL columns so the value in the `defdir` column is set to the value that you specified either in the asset descriptor file or in the **DefDir** field in the **Create Asset Table** form when you create the asset type.
   - If you have checked the **Add 'General' category** checkbox, Asset Maker adds one row to the `Category` table for the new asset type and names that category `General`.

4. Register the elements.

   When you register the elements, AssetMaker does the following:

   - Creates a subdirectory in the `ElementCatalog` table under `OpenMarket/Xcelerate/AssetType` directory for the new asset type.

- Copies elements from the `AssetStubCatalog` table to the new subdirectory in the `ElementCatalog` table. These elements render CS-Direct forms for working with assets of this type and provide the processing logic for the CS-Direct functions.
- Creates SQL statements that implement searches on individual fields in the search forms. These statements are placed in the `SystemSQL` table.

When you use CS-Direct to work on an asset of this type (create, edit, inspect, and so on), AssetMaker parses the asset descriptor file, which is now located in the `AssetType` table, and passes its values to CS-Direct so that the forms are specific to the asset type. Statements in the asset descriptor file determine the input types of the fields, specify field length restrictions, and determine whether the field is displayed on search and search results forms.

Note that after you create an asset type, there are several configuration steps to complete before you can use it; for example, enabling the asset type on the sites that need to use the asset type, creating Start Menu shortcuts, and so on.

The flow chart in Figure 5 summarizes how AssetMaker works, and which database tables are involved when a basic asset type is created.

**Figure 4:** Asset Types: Database Tables and CS-Direct Forms

When rendering the content-entry form below, Content Server displays the names of columns in the database table as field names in the form. Field names (specified by developers) define the asset type; field values (entered by content providers) define the asset.

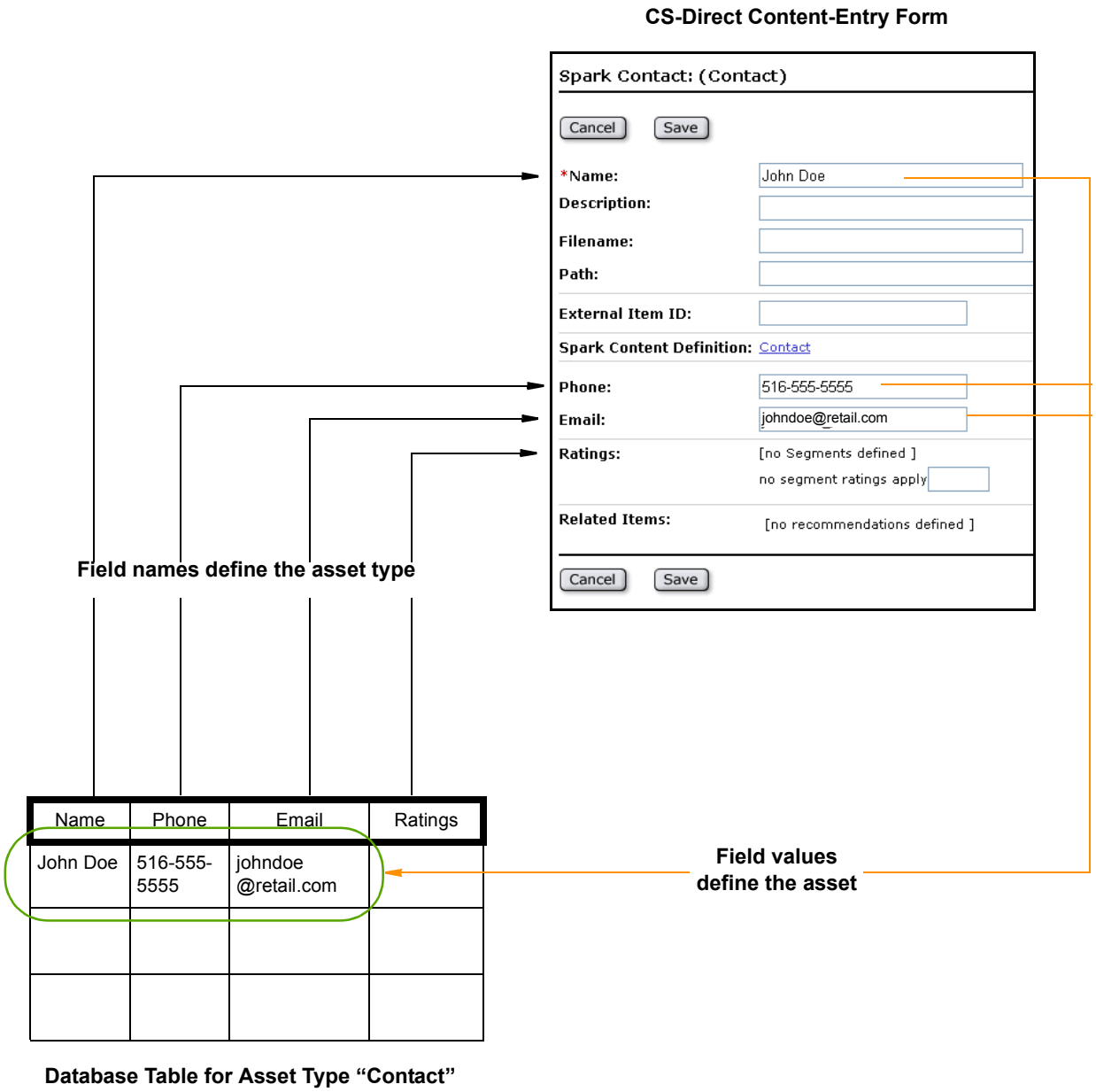
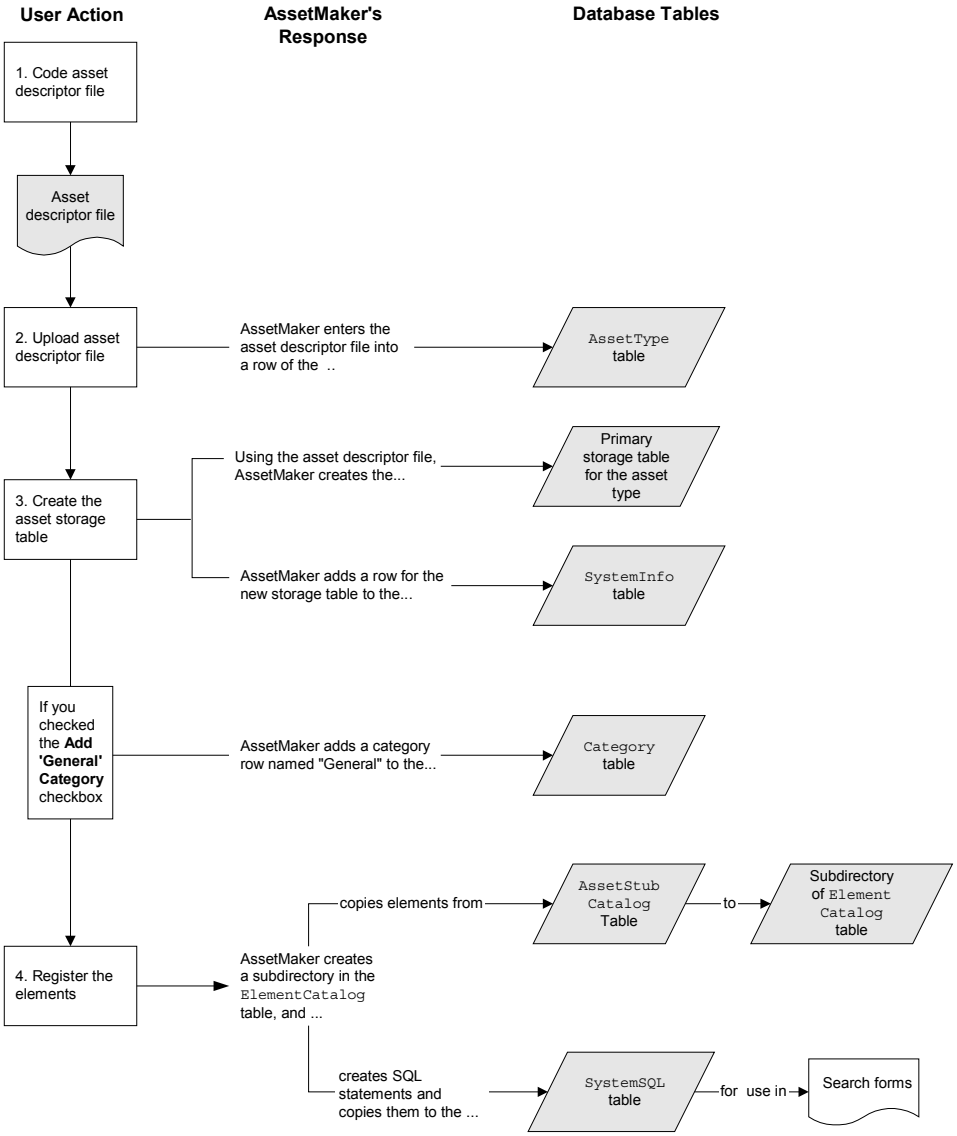


**Figure 5:** How AssetMaker Works



# Asset Descriptor Files

Using the AssetMaker XML tags, you code asset descriptor files that define the asset types you design for your systems.

## What Is an Asset Descriptor File?

An asset descriptor file is a valid XML document in which developers use AssetMaker tags to define a basic asset type. An asset descriptor file does the following two things:

- Describes the asset type in terms of data structure. It specifies the name of the database table, the names of the columns, the columns' data types, and the sizes of the fields on the CS-Direct forms.
- Formats the HTML forms that are displayed by CS-Direct when users work with assets of the given type. Formatting an HTML form means naming the fields on the form, displaying the fields in required format (for example, check box, radio button, or drop-down list), accounting for field specifications (such as the number of characters that can be entered in to a text field), and so on.

AssetMaker uses the asset descriptor file to create a database table for the new asset type. When content providers work with assets of the given type (create, edit, and so on), AssetMaker parses the asset descriptor file, using the data in the file to customize the forms that CS-Direct displays.

**Note**

For reference, sample AssetMaker descriptor code is provided on the Content Server installation medium, in the "Samples" folder. The same folder contains the `readme.txt` file that describes the sample descriptor files.

## Format and Syntax

The basic format for every asset descriptor file is shown below. To the right of each AssetMaker tag is a brief description of the tag.

```
<?xml version="1.0" ?>
<ASSET ...>                      Names the asset type (storage table)
<PROPERTIES>                     Starts the properties specification section

   <PROPERTY ...>                Specifies column and field name for the property
      <STORAGE .../>             Specifies data type for the column
      <INPUTFORM .../>           Specifies field format on New, Edit, Inspect forms
      <SEARCHFORM .../>          Specifies field format on Advanced Search form
      <SEARCHRESULTS .../>       Specifies which fields are shown in search results
   </PROPERTY>

   <PROPERTY ...>
      <STORAGE .../>
      <INPUTFORM .../>
      <SEARCHFORM .../>
      <SEARCHRESULTS .../>
   </PROPERTY>

   <PROPERTY ...>
   ...
```

```
   ...
</PROPERTIES>          Ends the properties specification section
</ASSET>               Ends the asset descriptor file
```

Shown next is the syntax of an asset descriptor file, indicating some of the parameters that an AssetMaker tag can take:

```
<?xml version="1.0" ?>
<ASSET NAME="assetTypeName" DESCRIPTION=""assetTypeName" ...>
<PROPERTIES>

   <PROPERTY NAME="fieldName1" DESCRIPTION="fieldName1"/>
         <STORAGE TYPE="VARCHAR" LENGTH="36"/>
         <INPUTFORM TYPE="TEXT" DESCRIPTION="fieldName1".../>
         <SEARCHFORM TYPE="TEXT" DESCRIPTION="fieldName1".../>
         <SEARCHRESULTS INCLUDE="TRUE"/>
   </PROPERTY>

   <PROPERTY NAME="fieldName2" DESCRIPTION="fieldName2"/>
         <STORAGE TYPE="INTEGER" LENGTH="4"/>
         <INPUTFORM TYPE="TEXT" DESCRIPTION="fieldName2".../>
         <SEARCHFORM TYPE="TEXT" DESCRIPTION="fieldName2".../>
         <SEARCHRESULTS INCLUDE="TRUE"/>
   </PROPERTY>

   .
   .
   .

</PROPERTIES>
</ASSET>
```

An overview of the tags in the asset descriptor file is given in this guide. Detailed information about the tags and their parameters, along with sample code, is given in the *Content Server Tag Reference*.

## Overview of the AssetMaker Tags

- An asset descriptor file begins with the standard XML version tag:

  `<?xml version="1.0" ?>`

- The `ASSET` tag, which follows the XML version tag, names the asset type and therefore its storage table in the Content Server database. The `ASSET` tag also sets some of the behavior and display attributes of assets of that type; for example, the `ASSET` tag determines what graphical notation designates that a field is required, and whether an asset can be previewed.

  The opening tag `<ASSET>` is always the first line of code and the closing tag `<\ASSET>` is always the last line of code in the asset descriptor file. Note that there is only one `ASSET` tag pair in each asset descriptor file because only one asset type per asset descriptor file can be created.

- The `PROPERTIES` tag marks the section of the file that holds the property descriptions. The opening tag `<PROPERTIES>` is always the second statement in the

asset descriptor file. There is only one PROPERTIES tag pair in each asset descriptor file.

**Note**

The PROPERTIES tag is required in every asset descriptor file, even if no PROPERTY tags are needed.

- The PROPERTY tags, nested within the PROPERTIES tag pair, specify the columns and fields for assets of this type. Each PROPERTY tag specifies the database name of the column that will hold the value(s) users will enter for this property, and the column's display name (i.e., its field name) as it will appear on the form that will be rendered for users who are to work with assets of this type.
- Nested inside each pair of PROPERTY tags are the following tags:
  - STORAGE—specifies the data type of the column that is being established by this property. Note that the data type in the STORAGE tag must map to one of the data types that is defined by the properties on the "Database" tab of the futuretense.ini file.
  - INPUTFORM—specifies the name and format of the field on the "New," "Edit," and "Inspect" forms. For example: Is the field a drop-down list or a check box or a text field? The field's input type must be compatible with the data type of the database column, as specified by the STORAGE statement.
  - SEARCHFORM—specifies the format of the field (property) when it appears on the "Advanced Search" form. If the SEARCHFORM statement is omitted from the PROPERTY section, the field being defined does not appear on the "Advanced Search" form.

    Note that if the value of the TYPE parameter is "Table" or "Date", a drop-down list will appear on the "Advanced Search" form for the asset type, but not on the SimpleSearch form.
  - SEARCHRESULTS—specifies which fields are displayed in the search results form after a search is run. The field value is also displayed if the INCLUDE parameter is set to "true". (This tag is optional.)

    If you are modifying a standard field, do not set SEARCHRESULTS to true for name or description.

For detailed information about these tags and their parameters, see the "AssetMaker Tags" chapter in the *Content Server Tag Reference*. That section also provides information about dependencies and restrictions among the parameters STORAGE TYPE, INPUTFORM TYPE, and SEARCHFORM TYPE.

## Columns in the Asset Type's Database Table

When AssetMaker creates the database table for a new asset type, it creates columns for all the properties defined by the PROPERTY tags in the asset descriptor file, and it creates default columns that are required by Content Server for its basic functionality. For a list of the default columns in each asset type's table, see "Default Columns in the Basic Asset Type Database Table" on page 191.

## The Source Column: A Special Case

All of the asset type tables can also have a `source` column. CS-Direct provides a `Source` table and a **Source** form on the **Admin** tab that you use to add the rows to the `Source` table. You can use this feature to identify where an asset originated. The Burlington Financial sample site, for example, uses `Source` to identify which wire feed service provided an article asset.

However, unlike the columns listed in the preceding table, the `source` column is not automatically created when AssetMaker creates the asset type table. To add the source column to your table and have it displayed on your asset forms, you must include a `PROPERTY` description for it in the asset descriptor file. For an example, see "Example: Adding the Source Column and Field" on page 283.

## Storage Types for the Columns

The `STORAGE TYPE` parameter specifies the data type of a column. The data types are defined by the Content Server database properties located in the `futuretense.ini` file.

The following table presents the possible data types for your asset type's table columns:

| Type (generic ODBC/JDBC data type) | Property |
|---|---|
| CHAR | cc.char |
| VARCHAR | cc.varchar |
| SMALLINT | cc.smallint |
| INTEGER | cc.integer |
| BIGINT | cc.bigint |
| DOUBLE | cc.double |
| TIMESTAMP | cc.datetime |
| BLOB | cc.blob |
| LONGVARCHAR | cc.bigtext |

## Input Types for the Fields

The `INPUT TYPE` parameter specifies how data can be entered in a field when it is displayed in the CS-Direct forms. The following table lists all the input types. Note that the input type for a field must be compatible with the data type of its column:

| Input TYPE | Description |
|---|---|
| TEXT | A single line of text.<br>Corresponds to the HTML input type named `TEXT`. |

| Input TYPE | Description |
|---|---|
| TEXTAREA | A text box, with scroll bars, that accepts multiple lines of text.<br>Corresponds to the HTML input type named TEXTAREA.<br>If you expect large amounts of text to be entered in the field, it is a good idea to create a text box that displays the contents of a URL column. To do so, you must specify a string for PROPERTY NAME that begins with the letters "url" and set the STORAGE TYPE to VARCHAR.<br>When a user clicks **Save**, the text entered into this kind of field is stored in the file directory specified as the default storage directory for this asset type. You can specify the default storage directory (defdir) in either the asset descriptor file, or in the AssetMaker form when you create the asset type.<br>**Note:**<br>• You can specify an unlimited size for a url field that is edited via a TEXTAREA field by not specifying a value for the MAXLENGTH parameter.<br>• If you are specifying a string for PROPERTY NAME that begins with the letters url, do not use the following suffixes: _type, _size, _folder, and _file. |
| UPLOAD | A field that takes a file name (a URL) and presents a **Browse** button so that you can either enter the path to and name of a file or browse to it and select it.<br>When you specify that a field is an upload field, set a string for PROPERTY NAME that begins with the letters "url" and set STORAGE TYPE (the property's data type) to VARCHAR.<br>You can also use the BLOB storage type for an upload field; in this case, the PROPERTY NAME string does not have to begin with url.<br>When the user clicks **Save**, Content Server uploads the selected file and stores it in the file directory specified as the default storage directory for this asset type. You can specify the default storage directory (defdir) in either the asset descriptor file, or in the AssetMaker form when you upload the file.<br>**Note:**<br>• The size of a file that is selected in an upload field cannot exceed 30 megabytes.<br>• If you are specifying a string for PROPERTY NAME that begins with the letters url, do not use the following suffixes: _type, _size, _folder, and _file. |
| SELECT | A field that presents a drop-down list of options that can be selected.<br>You can either specify the options that are presented in the list or you can specify a query so that the options are selected from the database (or an external table) and presented dynamically.<br>Corresponds to the HTML input type SELECT. |

| Input TYPE | Description |
|---|---|
| CHECKBOX | A check box field.<br>You can specify the names of the check box options or you can specify a query so that the names are selected from the database (or an external table) and presented dynamically. This input type allows the user to select more than one option.<br>Corresponds to the HTML input type CHECKBOX. |
| RADIO | A radio button control.<br>You can either specify the names of the radio options or you can specify a query so that the names are selected from the database (or an external table) and presented dynamically. This input type allows the user to select only one option.<br>Corresponds to the HTML input type RADIO. |
| EWEBEDITPRO | A field whose contents you edit by using the eWebEditPro HTML editor, a third-party tool from Ektron, Inc.<br>When you specify that a field is an eWebEditPro field, it's best if you make it a URL field. That is, set a string for PROPERTY NAME that begins with the letters "url" and set STORAGE TYPE (the property's data type) to VARCHAR.<br>Note that if you are specifying a string form PROPERTY NAME that begins with the letters "url," do not use the following suffixes: _type, _size, _folder, and _file. |
| ELEMENT | Calls an element that you create to display a field on the **ContentForm**, **ContentDetails**, or **SearchForm** forms. The custom element must be found at one of the following locations:<br>• For a field on the **ContentForm** form:<br>OpenMarket/Xcelerate/AssetType/*myAssetType*/ContentForm/*fieldname*<br>• For a field on the **ContentDetails** form:<br>OpenMarket/Xcelerate/AssetType/*myAssetType*/ContentDetails/*fieldname*<br>• For a field on the **SearchForm** form:<br>OpenMarket/Xcelerate/AssetType/*myAssetType*/SearchForm/*fieldname*<br>Where *myAssetType* is the asset type that you are creating the custom field for, and *fieldname* is the name of the custom field.<br>An ELEMENT field can have any storage type, including BLOB. |

## Datatypes for Standard Asset Fields

You can customize the appearance of Content Server's standard asset fields. All other changes are conditional on the type of field as described below:

- For all standard fields, you can change their display names.
- For a standard field that is not a system field, you must not change its datatype, with one exception: If the datatype is `VARCHAR`, you can change only its length.
- For system fields, you must not change the datatype (including the length of a `VARCHAR` type of field).

The following table lists the datatypes of standard fields (and indicates whether they are also system fields):

| Standard Field | System Field | Datatype |
|---|---|---|
| ID | ✓ | NOT NULL NUMBER(38) |
| NAME | | NOT NULL VARCHAR(64) |
| DESCRIPTION | | VARCHAR(128) |
| TEMPLATE | ✓ | VARCHAR(64) |
| SUBTYPE | | VARCHAR(24) |
| FILENAME | | VARCHAR(64) |
| PATH | | VARCHAR(255) |
| STATUS | ✓ | NOT NULL VARCHAR(2) |
| EXTERNALDOCTYPE | ✓ | VARCHAR(64) |
| URLEXTERNALDOCXML | ✓ | VARCHAR(255) |
| URLEXTERNALDOC | ✓ | VARCHAR2(255) |
| CREATEDBY | ✓ | NOT NULL VARCHAR(64) |
| UPDATEDBY | ✓ | NOT NULL VARCHAR(64) |
| CREATEDDATE | ✓ | NOT NULL DATE |
| UPDATEDDATE | ✓ | NOT NULL DATE |
| STARTDATE | | DATE |
| ENDDATE | | DATE |

## Elements and SQL Statements for the Asset Type

After you upload an asset descriptor file, you "register" the elements. When you register elements, AssetMaker copies elements in the `AssetStubElementCatalog` table to a directory in the `ElementCatalog` table for this asset type.

Additionally, AssetMaker copies several SQL statements that implement the CS-Direct searches on the **Simple Search** and the **Advanced Search** forms for assets of this type.

If necessary, you can customize the SQL statements, the asset type-specific elements, or, in some cases, the elements in the `AssetStubElementCatalog` table.

**Caution**

**Under no circumstances** should you modify any of the other CS-Direct elements.

For information about customizing your elements, see "Step 7: (Optional) Customize the Asset Type Elements" on page 292.

## The Elements

AssetMaker places the elements for your new asset type to the `ElementCatalog` table according to the following naming convention:

`OpenMarket/Xcelerate/AssetType/`*`YourNewAssetType`*

For example, the elements for the sample asset type "ImageFile" are located here:

`OpenMarket/Xcelerate/AssetType/ImageFile`

The following table lists the elements that AssetMaker copies for each asset type:

| Element | Description |
|---|---|
| `ContentForm` | Renders the **New** and **Edit** forms for assets of this type.<br>When the function is invoked, AssetMaker uses the `INPUTFORM` statements in the asset descriptor file to format these forms. |
| `ContentDetails` | Formats the **Inspect** form for assets of this type.<br>When the function is invoked, AssetMaker uses the `INPUTFORM` statements in the asset descriptor file to customize these forms. |
| `SimpleSearch` | Renders the **Simple Search** form for assets of this type.<br>When the function is invoked, AssetMaker uses the `SEARCHFORM` statements in the asset descriptor file to format these forms. |
| `SearchForm` | Formats the **Advanced Search** form for assets of this type.<br>When the function is invoked, AssetMaker uses the `SEARCHFORM` statements in the asset descriptor file to format these forms. |
| `AppendSelectDetails` | Builds the SQL queries on the individual fields in the **Advanced Search** form.<br>When the **Advanced Search** form is rendered, AssetMaker uses the `SEARCHFORM` statements in the asset descriptor file to customize the form. |

| Element | Description |
|---|---|
| `AppendSelectDetailsSE` | Builds the SQL queries on the individual fields in the **Advanced Search** form when your system is using an external search engine.<br>When this function is invoked, AssetMaker uses the `SEARCHFORM` statements in the asset descriptor file to create the SQL queries. |
| `IndexAdd` | The `IndexAdd` and `IndexReplace` elements establish which fields (columns) are indexed by the search engine when you are using a search engine. By default, only the standard fields are indexed. If you want other fields indexed, you must customize these forms. For more information, see "Step 7: (Optional) Customize the Asset Type Elements" on page 292. |
| `IndexReplace` | See the description of `IndexAdd`, above. |
| `Tile` | Formats the **Search Results** page, a page that lists the assets that meet the search criteria, for assets of this type.<br>When the page is rendered, AssetMaker uses the `SEARCHRESULTS` statements in the asset descriptor file to display the results. |
| `LoadTree` | Determines how assets of this type appear when they are displayed on any tab in the tree **other than** the **Site Plan** tab. |
| `LoadSiteTree` | Determines how assets of this type appear when they are displayed on the **Site Plan** tab. |
| `PreUpdate` | Is called before a function that writes to the database is completed. In other words, before an asset is saved and during the create, edit, delete, or XMLPost functions, this element is called.<br>This element takes no input from the asset descriptor file. However, you can customize it directly. |
| `PostUpdate` | Is called after a function that writes to the database is completed. In other words, after an asset is created, edited, deleted, or imported with XMLPost, this element is called.<br>You can customize this element. |

## The SQL Statements

AssetMaker places the SQL statements in the `SystemSQL` table according to the following naming convention:

`OpenMarket/Xcelerate/AssetType/`*`YourNewAssetType`*

For example, the elements for the sample asset type “ImageFile” are located here:

`OpenMarket/Xcelerate/ImageFile`

The following table lists the SQL elements that AssetMaker creates:

| Statement | Description |
|---|---|
| `SelectSummary` | A SQL statement that defines the query used in the **Simple Search** and **Advanced Search** form for assets of this type. |
| `SelectSummarySE` | Not used. |

# Creating Basic Asset Types

The length of time that it takes you to create a new asset type can range widely depending on the complexity of your asset type.

A simple asset type might require you to code one simple asset descriptor file and then upload it. A more complicated asset type might require you to modify the code in the elements that AssetMaker creates for your asset type or to add a database table to hold information that you want displayed in a drop-down list.

## Overview

Following is an overview of the process for creating and configuring a new asset type. This chapter describes each of the steps, except as noted:

1. Code an asset descriptor file (page 280).
2. Upload the asset descriptor file, using AssetMaker in Content Server's **Admin** tab (page 288).
3. Create the database table and register the asset type elements by copying the asset type elements from the `AssetStubElementCatalog` table to the appropriate directory in the `ElementCatalog` table (page 289).
4. Configure the asset type (page 290).
5. Enable the asset type for the site that you are using to develop assets on and create a **Start Menu** shortcut so that you can work with the asset type (page 291).
6. Examine the **New**, **Edit**, **Inspect**, **Search**, and **Search Results** forms. If necessary, fine-tune the asset descriptor file, and re-register the asset type elements (page 291).
7. (Optional) If necessary, customize the asset type elements (page 292).
8. (Optional) Add **Subtype** entries for the new asset type (page 295).
9. (Optional) Create asset **Association** fields for the new asset type (page 296).
10. (Optional) Add **Category** entries for the new asset type (page 298).
11. (Optional) Add **Source** entries for the new asset type (page 299).
12. (Optional) Add **Mimetypes** for the new asset type (page 300).
13. (Optional) If you are using a search engine rather than the Content Server database search utility to perform the logic behind the search forms and you want to use it on your new asset type, edit your search elements to enable indexed searching (page 301).
14. Create and assign the icon that will represent the asset type in the Dash interface. The icon will be displayed in the "Type" column in the list of returned assets whenever a user performs a search or runs a tag (page 301).
15. Code templates for assets of this type (page 302, which refers you to Chapter 27, "Coding Elements for Templates and CSElements").
16. Move the asset types to the other systems, management and delivery (page 302). This allows your administrator to complete the final steps in creating the asset type, including setting up workflow and creating start menu items.

# Before You Begin

Before you begin coding your asset descriptor files, you must plan your design and set up your development system, as described in the following sections.

## Plan the Asset Type Design

Be sure to design your asset types on paper before you start coding an asset descriptor file. Consider the following kinds of details:

- What fields do you need?

  In general, try to minimize the number of fields that you use by organizing the information into useful units. When determining those units, consider both the information you plan to display on your online site and the data-entry needs of the content providers who will enter that data.
- What is the appropriate data type for each field?
- For fields with options, how will you supply the options? With a static list coded in the asset descriptor file or with a lookup table that holds the valid options?
- Which Content Server features will you use to organize or categorize assets of this type? For example, source, category, and asset associations. For each one, determine its name and plan how it will be used both on the management system and in the design of your online site.
- Does the implementation of your site design require assets of this type to use a different default template based on the publishing target that they are published to? If so, you will need to use the **Subtype** feature. Determine the names of the subtypes that you will need for assets of this type.

## Set Up Your Development System

Also before you begin, be sure to set up your development system. For information about any of these preliminary steps, see the *Content Server Administrator's Guide*:

- Create the appropriate sites.
- Create a user name for yourself that has administrator rights and enable that user name on all of the sites on your development system. (Be sure that the TableEditor ACL is assigned to your user name or you will be unable to create new asset types.)

**Note**

Without administrator rights, you do not have access to the **Admin** tab, which means that you cannot perform any of the procedures in this chapter. For the sake of convenience, assign the **Designer** and **GeneralAdmin** roles to your user name. That way you will have access to all the tabs and all of the existing **Start Menu** shortcuts for the assets in the sample site.

- If you plan to use eWebEditPro, a third-party HTML editor, you must configure it on the systems that you plan to use it on. It is not delivered with CS-Direct (or CS-Direct Advantage).

## Step 1: Code the Asset Descriptor File

As described in "Asset Descriptor Files" on page 268, this is the basic format of an asset descriptor file:

```
<?xml version="1.0" ?>
<ASSET NAME="assetName"...>
       <PROPERTIES>
          <PROPERTY.../>
               <STORAGE.../>
               <INPUTFORM.../>
               <SEARCHFORM.../>
               <SEARCHRESULTS.../>
          </PROPERTY>
            <PROPERTY... />
               <STORAGE.../>
               <INPUTFORM.../>
               <SEARCHFORM.../>
               <SEARCHRESULTS.../>
          </PROPERTY>
         </PROPERTIES>
</ASSET>
```

To code your asset descriptor files, read the "AssetMaker Tags" chapter in the *Content Server Tag Reference* and use the tags described in that chapter to code the file. You can use the native XML editor in Content Server Explorer to code the file or you can use any other XML editor.

Note that you can customize the appearance of standard asset fields by including them in your asset descriptor file. Changing a field's storage type is conditional. For example, if the field is a system field, its storage type must not be changed. For the list of standard fields, their storage types, and allowed changes to storage type, see "Datatypes for Standard Asset Fields" on page 274.

This section offers a sample asset descriptor file and several examples about coding specific kinds of properties.

### Sample Asset Descriptor File: ImageFile.xml

If the Burlington Financial sample site is installed on your system, you will find the `ImageFile.xml` asset descriptor file in the `AssetType` table. You can either start Content Server Explorer and open the file or you can examine it here:

```
<!-- this is the description of an asset -->
<ASSET NAME="ImageFile" DESCRIPTION="ImageFile" MARKERIMAGE="/
Xcelerate/data/help16.gif" PROCESSOR="4.0"
DEFDIR="c:\FutureTense\Storage\ImageFile">
    <PROPERTIES>

        <PROPERTY NAME="source" DESCRIPTION="Source">
            <STORAGE TYPE="VARCHAR" LENGTH="24"/>
            <INPUTFORM DESCRIPTION="Source" TYPE="SELECT"
TABLENAME="Source" OPTIONDESCKEY="description"
OPTIONVALUEKEY="source" SOURCETYPE="TABLE"/>
```

```
            <SEARCHFORM DESCRIPTION="Source" TYPE="SELECT"
TABLENAME="Source" OPTIONDESCKEY="description"
OPTIONVALUEKEY="source" SOURCETYPE="TABLE"/>
       </PROPERTY>
        <PROPERTY NAME="urlpicture" DESCRIPTION="Image File">
            <STORAGE TYPE="VARCHAR" LENGTH="255"/>
            <INPUTFORM TYPE="UPLOAD" WIDTH="36" REQUIRED="NO"
LINKTEXT="Image"/>
        </PROPERTY>

        <PROPERTY NAME="urlthumbnail" DESCRIPTION="Thumbnail
File">
            <STORAGE TYPE="VARCHAR" LENGTH="255"/>
            <INPUTFORM TYPE="UPLOAD" WIDTH="36" REQUIRED="NO"
LINKTEXT="Image"/>
        </PROPERTY>

       <PROPERTY NAME="mimetype" DESCRIPTION="Mimetype">
            <STORAGE TYPE="VARCHAR" LENGTH="36"/>
            <INPUTFORM TYPE="SELECT" SOURCETYPE="TABLE"
TABLENAME="MimeType" OPTIONDESCKEY="description"
OPTIONVALUEKEY="mimetype"
      SQL="SELECT mimetype, description FROM MimeType WHERE
keyword = 'image' AND isdefault = 'y'" INSTRUCTION="Add more
options to mimetype table with isdefault=y and keyword=image"/>
            <SEARCHFORM DESCRIPTION="MimeType" TYPE="SELECT"
SOURCETYPE="TABLE" TABLENAME="MimeType"
OPTIONDESCKEY="description" OPTIONVALUEKEY="mimetype"
      SQL="SELECT mimetype, description FROM MimeType WHERE
keyword = 'image' AND isdefault = 'y'"/>
        </PROPERTY>

        <PROPERTY NAME="width" DESCRIPTION="Width">
            <STORAGE TYPE="INTEGER" LENGTH="4"/>
            <INPUTFORM TYPE="TEXT" WIDTH="4" MAXLENGTH="4"
REQUIRED="NO" DEFAULT=""/>
            <SEARCHFORM DESCRIPTION="Width is" TYPE="TEXT"
WIDTH="4" MAXLENGTH="4" VERB="="/>
        </PROPERTY>

        <PROPERTY NAME="height" DESCRIPTION="Height">
            <STORAGE TYPE="INTEGER" LENGTH="4"/>
            <INPUTFORM TYPE="TEXT" WIDTH="4" MAXLENGTH="4"
REQUIRED="NO" DEFAULT=""/>
            <SEARCHFORM DESCRIPTION="Height is" TYPE="TEXT"
WIDTH="4" MAXLENGTH="4" VERB="="/>
        </PROPERTY>

        <PROPERTY NAME="align" DESCRIPTION="Alignment">
            <STORAGE TYPE="VARCHAR" LENGTH="8"/>
```

```
            <INPUTFORM TYPE="SELECT" SOURCETYPE="STRING"
OPTIONVALUES="Left,Center,Right"
OPTIONDESCRIPTIONS="Left,Center,Right"/>
            <SEARCHFORM DESCRIPTION="Alignment" TYPE="SELECT"
SOURCETYPE="STRING" OPTIONVALUES="Left,Center,Right"
OPTIONDESCRIPTIONS="Left,Center,Right"/>
        </PROPERTY>

        <PROPERTY NAME="artist" DESCRIPTION="Artist">
            <STORAGE TYPE="VARCHAR" LENGTH="64"/>
            <INPUTFORM TYPE="TEXT" WIDTH="36" MAXLENGTH="36"
REQUIRED="NO" DEFAULT=""/>
            <SEARCHFORM DESCRIPTION="Artist contains" TYPE="TEXT"
WIDTH="36" MAXLENGTH="64"/>
        </PROPERTY>

        <PROPERTY NAME="alttext" DESCRIPTION="Alt Text">
            <STORAGE TYPE="VARCHAR" LENGTH="255"/>
            <INPUTFORM TYPE="TEXT" WIDTH="48" MAXLENGTH="255"
REQUIRED="NO" DEFAULT=""/>
            <SEARCHFORM DESCRIPTION="Alt Text contains"
TYPE="TEXT" WIDTH="48" MAXLENGTH="255"/>
        </PROPERTY>

        <PROPERTY NAME="keywords" DESCRIPTION="Keywords">
            <STORAGE TYPE="VARCHAR" LENGTH="128"/>
            <INPUTFORM TYPE="TEXT" WIDTH="48" MAXLENGTH="128"
REQUIRED="NO" DEFAULT=""/>
           <SEARCHFORM DESCRIPTION="Keywords contain" TYPE="TEXT"
WIDTH="48" MAXLENGTH="128"/>
        </PROPERTY>

        <PROPERTY NAME="imagedate" DESCRIPTION="Image date">
            <STORAGE TYPE="TIMESTAMP" LENGTH="8"/>
            <INPUTFORM TYPE="ELEMENT" WIDTH="24" MAXLENGTH="48"
REQUIRED="NO" DEFAULT="" INSTRUCTION="Format: yyyy-mm-dd hh:mm"/>
                       <SEARCHFORM DESCRIPTION="Image date"
TYPE="ELEMENT" WIDTH="48" MAXLENGTH="128"/>
        </PROPERTY>

    </PROPERTIES>
</ASSET>
```

Examine this asset descriptor file and then, if Burlington Financial is installed on your system, start CS-Direct, select the Burlington Financial site, examine the CS-Direct forms for the imagefile asset type, and compare the forms to the asset descriptor file.

Note the following about the `ImageFile` asset descriptor file:

- The `ASSET` tag provides a value for the `DEFDIR` parameter. All asset tables have at least two URL columns (upload fields) by default, which means you must set a value for the default storage directory (defdir) of any new asset type. (the imagefile asset type has two additional URL columns: `urlpicture` and `urlthumbnail`.)

You can set the `defdir` value either with the `ASSET` tag's `DEFDIR` parameter, or with the **defdir** field in the AssetMaker **Create Asset Table** form.

> **Note**
>
> A `defdir` set using the **Create Asset Table** form overrides a defdir set in the asset descriptor file.

For more information about URL columns, see "Indirect Data Storage with the Content Server URL Field" on page 219.

- There are no `PROPERTY` statements for any of the default columns that AssetMaker creates in an asset type's database table. (Those columns are listed in "Default Columns in the Basic Asset Type Database Table" on page 191.)
- Not every property that has a `SEARCHFORM` statement has a matching `SEARCHRESULTS` statement. In other words, if you decide to put a field on the **Advanced Search** form, it does not mean that you have to display the data from that field on the **Search Results** form.

## Example: Adding the Source Column and Field

The source column is not created by default even though CS-Direct has a Source feature on the **Admin** tab. In order to use the Source feature on your new asset types, you must include a property statement for the source column and field.

Note the following:

- `STORAGE TYPE` must be set to `"VARCHAR"` and `LENGTH` must be set to `"24"`.
- `INPUTFORM SOURCETYPE` must be set to `"TABLE"` and `TABLENAME` must be set to `"Source"`.

For example:

```
<PROPERTY NAME="source" DESCRIPTION="Source">
   <STORAGE TYPE="VARCHAR" LENGTH="24"/>
   <INPUTFORM TYPE="SELECT" TABLENAME="Source"
   OPTIONDESCKEY="description" OPTIONVALUEKEY="source"
   SOURCETYPE="TABLE"/>
   <SEARCHFORM DESCRIPTION="Source" TYPE="SELECT"
   TABLENAME="Source" OPTIONDESCKEY="description"
   OPTIONVALUEKEY="source" SOURCETYPE="TABLE"/>
</PROPERTY>
```

## Example: An Asset Type With Unnamed Associations

You can create an asset type that supports unnamed associations (multiple asset types associated with the asset) by setting the `ASSET` tag's `UNNAMEDASSOCIATIONS` parameter to `YES`. This causes a **Contents** field to appear in the asset's ContentForm, similar to the **Contains** field on the Page asset forms.

The sample code for creating an asset type with unnamed associations follows:

```
<ASSET NAME="Container" DESCRIPTION="Container"
PLURAL="Containers" UNNAMEDASSOCIATIONS="YES" DEFDIR="C:/
FutureTense/Storage/Container">
<PROPERTIES>
```

```
</PROPERTIES>
</ASSET>
```

## Upload Example 1: A Standard Upload Field

To create an upload field with a **Browse** button, code the `PROPERTY` statement as follows:

1. The string set for `PROPERTY NAME` must begin with the letters `url`.
2. The value for `STORAGE TYPE` must be set to `VARCHAR`.
3. The value for `INPUT TYPE` must be set to `UPLOAD`.

Here is a code snippet of an upload field from the `ImageFile` asset descriptor file:

```
<PROPERTY NAME="urlpicture" DESCRIPTION="Image File">
   <STORAGE TYPE="VARCHAR" LENGTH="255"/>
   <INPUTFORM TYPE="UPLOAD" WIDTH="36" REQUIRED="NO"
   LINKTEXT="Image"/>
</PROPERTY>
```

**Note**

The size of a file that you can select in an upload field is limited to 30 megabytes.

## Upload Example 2: A Text Box Field

To create an upload field with a text box that you can enter the text in (rather than with a **Browse** button that you use to select a file), code the `PROPERTY` statement as follows:

1. The string set for `PROPERTY NAME` must begin with the letters `url`.
2. The value for `STORAGE TYPE` must be set to `VARCHAR`.
3. The value for `INPUT TYPE` must be set to `TEXTAREA`.

The following code snippet creates a text area field for a `url` column:

```
<PROPERTY NAME="urlbody" DESCRIPTION="Article Body">
   <STORAGE TYPE="VARCHAR" LENGTH="256"/>
        <INPUTFORM TYPE="TEXTAREA" COLS="48" ROWS="25"
    REQUIRED="YES"/>
</PROPERTY>
```

## Upload Example 3: An eWebEdit Pro Field

eWebEditPro is a third-party HTML editor from Ektron, Inc. For information about configuring your system to use eWebEditPro, see "Configuring eWebEditPro" on page 404.

Code the property statement as follows:

1. The `PROPERTY NAME` should begin with the letters "`url`". In other words, you should use a URL column for the field. If you do not, you run the risk of making your field too small.
2. The value for `STORAGE TYPE` must be set to `VARCHAR`.

3. `INPUT TYPE` must be set to `EWEBEDITPRO`.

For example:

```
<PROPERTY NAME=“urlbody” DESCRIPTION=“Body”>
   <STORAGE TYPE=“VARCHAR” LENGTH=“500”/>
   <INPUTFORM TYPE=“EWEBEDITPRO” WIDTH=“300” HEIGHT=“300”
   REQUIRED=“YES” INSTRUCTION=“Be concise! No more than 3
   paragraphs.”/>
</PROPERTY>
```

## Upload Example 4: A Text Field With Embedded Links

You can allow content editors to embed hyperlinks within a text field. If embedded links are enabled for a text field, two embedded link buttons—**Add Link** and **Include**—appear next to the field. To enable embedded links for a text field, code the property statement as follows:

1. The string set for `PROPERTY NAME` must begin with the letters `url`.
2. The value for `STORAGE TYPE` must be set to `VARCHAR`.
3. The value for `INPUT TYPE` must be set to `TEXTAREA`.

The following code snippet creates a text area field for a `url` column:

```
<PROPERTY NAME="urltext" DESCRIPTION="Text">
   <STORAGE TYPE="VARCHAR" LENGTH="2000"/>
        <INPUTFORM TYPE="TEXTAREA" COLS="48" ROWS="25"
EMBEDDEDLINKS=”YES” REQUIRED="YES"/>
</PROPERTY>
```

## Upload Example 4: A Field That Uploads a Binary File

The following code creates a field where you can upload a blob. Note that if you do not specify the `MIMETYPE`, you may not be able to view the blob from the **Edit** and **Inspect** forms.

```
<PROPERTY NAME="type_binary" DESCRIPTION="Binary">
         <STORAGE TYPE="BINARY"/>
         <INPUTFORM TYPE="UPLOAD" WIDTH="24" MAXLENGTH="64"
MIMETYPE="application/msword" LINKTEXT="Edit with Microsoft Word"
INSTRUCTION="maps to cc.blob"/>
    </PROPERTY>
```

## Example: Enabling path, filename, startdate, and enddate

The path, filename, startdate, and enddate columns are special cases.

AssetMaker creates columns for path, filename, startdate, and enddate without requiring a `PROPERTY` statement for them. However, while these columns exist, their fields do not appear on your asset forms unless you include a `PROPERTY` statement for them in the asset descriptor file.

Note the following about these columns:

- For **path**, `STORAGE TYPE` must be set to `VARCHAR` and `LENGTH` must be set to `255`.
- For **filename**, `STORAGE TYPE` must be set to `VARCHAR` and `LENGTH` must be set to `128`.

- For **startdate**, `STORAGE TYPE` must be set to `TIMESTAMP`.
- For **enddate**, `STORAGE TYPE` must be set to `TIMESTAMP`.

**Note**

If you include one of these standard columns in your asset descriptor file but your storage type does not match the one specified in this list, AssetMaker cannot create the asset type.

For example:

```
<PROPERTY NAME="path" DESCRIPTION="Path">
   <STORAGE TYPE="VARCHAR" LENGTH="255"/>
   <INPUTFORM DESCRIPTION="Path" TYPE="TEXT" LENGTH="255"/>
</PROPERTY>
```

## Example: Using a Query to Obtain Options for a Drop-Down List

When the `INPUTFORM TYPE` of your property is `SELECT`, you can have CS-Direct populate the drop-down list for the select field with a static list of items that you provide with the `OPTIONDESCRIPTIONS` parameter, or with a list of items that CS-Direct obtains, dynamically, from a database table.

Another example of a select field that populates its drop-down list dynamically from a table is the Mimetype field on the imagefile forms, which queries the MimeType table for its options. Here's the code:

```
<PROPERTY NAME="mimetype" DESCRIPTION="Mimetype">

   <STORAGE TYPE="VARCHAR" LENGTH="36"/>

   <INPUTFORM TYPE="SELECT" SOURCETYPE="TABLE"
   TABLENAME="mimetype" OPTIONDESCKEY="description"
   OPTIONVALUEKEY="mimetype" SQL="SELECT mimetype, description
   FROM mimetype WHERE keyword = 'image' AND
   isdefault = 'y'" INSTRUCTION="Add more options to mimetype
   table with isdefault=y and keyword=image"/>

   <SEARCHFORM DESCRIPTION="Mimetype" TYPE="SELECT"
   SOURCETYPE="TABLE" TABLENAME="mimetype"
   OPTIONDESCKEY="description" OPTIONVALUEKEY="mimetype"
   SQL="SELECT mimetype, description FROM mimetype WHERE keyword =
   'image' AND isdefault = 'y'"/>

</PROPERTY>
```

This example shows a field that not only selects items from a database table, but, through an additional SQL query, further restricts which items are returned from that table, as well.

## Example: Using a Query to Obtain Labels for Radio Buttons

If the `INPUTFORM TYPE` of your property is `RADIO`, you can input the label for each radio button using a static list of items that you provide with the `OPTIONDESCRIPTIONS` parameter, or with a list of items that CS-Direct obtains, dynamically, from a database table.

The following sample code creates radio buttons with labels drawn from the `CreditCard` table:

```
<PROPERTY NAME="sqlrbcc" DESCRIPTION="SQL RB Credit Card">
            <STORAGE TYPE="VARCHAR" LENGTH="4"/>
            <INPUTFORM TYPE="RADIO" SOURCETYPE="TABLE"
TABLENAME="CreditCard" RBVALUEKEY="ccvalue"
RBDESCKEY="ccdescription" />
      <SEARCHFORM TYPE="SELECT" DESCRIPTION="Credit Card:"
SOURCETYPE="TABLE" TABLENAME="CreditCard" OPTIONVALUEKEY="ccvalue"
OPTIONDESCKEY="ccdescription" SQL="SELECT ccvalue,ccdescription
FROM CreditCard ORDER BY ccdescription"/>
        </PROPERTY>
```

## Example: Creating a Field with the ELEMENT Input Type

You can modify the fields on your asset forms by using the `ELEMENT` input type to call custom code to display the fields as you want them. You can use this method to create new asset fields, or to change the appearance of standard asset fields—though you cannot modify the storage type of a standard asset field.

An `ELEMENT` field can have any storage type, including `BLOB`.

When you set a field's input type to `ELEMENT`, Content Server calls a custom element to display the field. The custom element must be found at one of the following locations:

- For a field on the **ContentForm** form:

  `OpenMarket/Xcelerate/AssetType/`*`myAssetType`*`/ContentForm/`*`fieldname`*

- For a field on the **ContentDetails** form:

  `OpenMarket/Xcelerate/AssetType/`*`myAssetType`*`/ContentDetails/`*`fieldname`*

- For a field on the **SearchForm** form:

  `OpenMarket/Xcelerate/AssetType/`*`myAssetType`*`/SearchForm/`*`fieldname`*

Note that *`myAssetType`* is the asset type that you are creating the custom field for, and *`fieldname`* is the name of the custom field.

The following exerpt from an asset descriptor file uses the `ELEMENT` input type:

```
 <PROPERTY NAME="imagedate" DESCRIPTION="Image date">
            <STORAGE TYPE="TIMESTAMP" LENGTH="8"/>
            <INPUTFORM TYPE="ELEMENT" WIDTH="24" MAXLENGTH="48"
REQUIRED="NO" DEFAULT="" INSTRUCTION="Format: yyyy-mm-dd hh:mm"/>
            <SEARCHFORM DESCRIPTION="Image date" TYPE="ELEMENT"
WIDTH="48" MAXLENGTH="128"/>
 </PROPERTY>
```

Note that the input form uses a customized field, but the search form and content details forms display default fields.

The following code excerpt is the element that the descriptor file calls:

```
<!-- OpenMarket/Xcelerate/AssetType/ImageFile/ContentForm/
imagedate
-
- INPUT
-     Variables.AssetType
      Variables.fieldname
      Variables.fieldvalue- default or value for this field
      Variables.datatype - from STORAGE tag in ADF for this field
      Variables.helpimage - help icon
      Variables.alttext - help text from INPUT tag in ADF
      Other fields from input tag are in:
         Variables.assetmaker/property/Variables.fieldname/
inputform/[tag attribute]
-  field name used in form should be
Variables.AssetType:Variables.fieldname

- OUTPUT
-
-->
   Enter date in the format yyyy-mm-dd hh:mm:ss<br/>

   <setvar NAME="inputfieldsize" VALUE="Variables.assetmaker/
property/Variables.fieldname/inputform/width"/>

   <callelement NAME="OpenMarket/Xcelerate/Scripts/FormatDate"/>
   <INPUT TYPE="text" SIZE="Variables.inputfieldsize"
NAME="Variables.AssetType:Variables.fieldname"
VALUE="Variables.fieldvalue"
REPLACEALL="Variables.inputfieldsize,Variables.fieldvalue,
Variables.fieldname,Variables.AssetType"

onChange="padDate(this.form.elements['Variables.AssetType:Variable
s.fieldname'].value,this,'Variables.AssetType:Variables.fieldname'
);"/>


</FTCS>
```

Note that you can customize as many fields as you want using the `ELEMENT` input type, but that you must write a separate element for each field that you want to modify.

## Step 2: Upload the Asset Descriptor File to Content Server

After you have coded the asset descriptor file for your asset type, use AssetMaker to upload it and register the new elements:

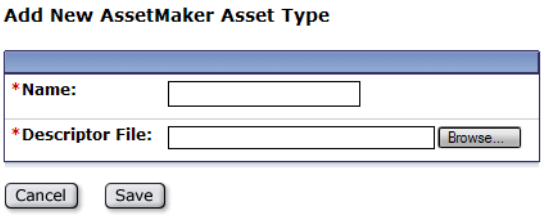
1. Log in to Content Server's Advanced interface as a general administrator (`fwadmin`/`xceladmin` are the default credentials):

   `http://<your_server>/Xcelerate/LoginPage.html`

   Note that you must have administrator rights associated with your user name (login name) in order to have access to AssetMaker.

2. Select the **Admin** tab > **AssetMaker** > **Add New**.

   The "Add New AssetMaker Asset Type" form appears:




3. Click in the **Name** field and enter the name of the new asset type. The string that you enter into this field must exactly match the string specified by the `ASSET NAME` parameter in the asset descriptor file that you are going to upload.
4. Click the **Browse** button next to the **Descriptor File** field and select the asset descriptor file.
5. Click **Save**.

   AssetMaker enters the file into the `AssetType` table (that is, it uploads the asset descriptor file to the default storage directory for the `AssetType` table), and then displays the "Asset Type" form:

## Step 3: Create the Asset Table

This step continues from step 5, above.

6. Select **Create Asset Table**.
7. Examine the value in the **Defdir** field and change it if necessary. AssetMaker reads this value from the asset descriptor file. You must enter a value in this field if either of the following conditions exist:
   - If you did not provide a value with the `DEFDIR` parameter for the `ASSET` tag in the asset descriptor.

- If you want to change the default storage directory, which is typical when you are migrating the asset type to another system.

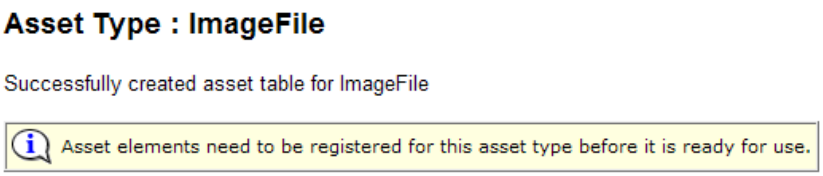
If you check the **Add 'General' category** checkbox, AssetMaker adds one row to the `Category` table for the new asset type and names that category **General**.

8. Click **Create Asset Table**.

   AssetMaker creates the table.

9. Select **Register Asset Elements** and then click the **Register Asset Elements** button.

   AssetMaker copies the elements from the `AssetStubElementCatalog` table to the asset type's directory in the `ElementCatalog` table and copies the SQL statements in the SystemSQL table. When it is finished, it displays a confirmation like this one:



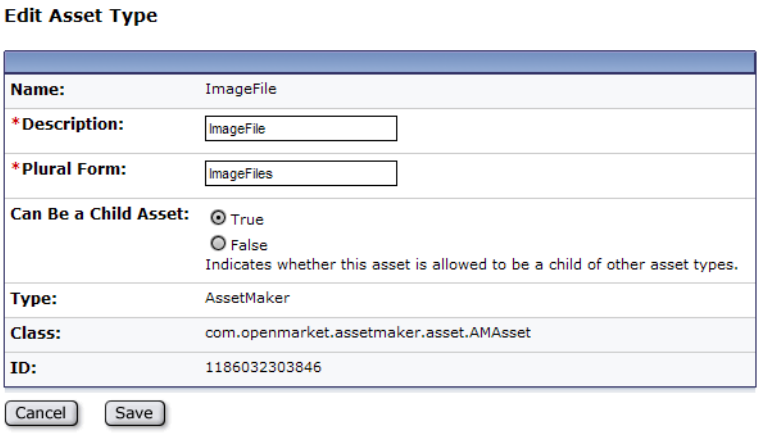

## Step 4: Configure the Asset Type

When AssetMaker created the new asset type (in step 2), it also created an icon and administrative forms for configuring the new asset type, located on the **Admin** tab.

Complete the following configuration steps:

1. On the **Admin** tab, expand the **Asset Types** icon.
2. Under **Asset Types**, select your new asset type. (If you do not see it in the list, click the right mouse button, select **Refresh** from the pop-up menu, and then select your new asset type.)
3. Click **Edit**.

   The following form appears:



4. (Optional) Click in the **Description** field and change it, if necessary. The text in this field is the name that CS-Direct uses for this asset type on the forms and lists in the Content Server interface. By default, **Description** is set to the value of the `ASSET DESCRIPTION` statement in the asset descriptor file.
5. (Optional) Click in the **Plural Form** field and change it, if necessary. The text in this field is the text that CS-Direct uses in the Content Server interface when it is appropriate to refer to the asset type in the plural. By default, **Plural Form** is set to the value of the `ASSET DESCRIPTION` statement in the asset descriptor file plus the letter "s." You can also specify your own plural form in the asset descriptor file by setting the value of the `ASSET` tag's `PLURAL` parameter.
6. (Optional) Click in the **Can Be Child Asset** field and change the value, if necessary. By default, this field is set to **True**, which means that this asset type can be the child asset type in an association field for another asset type. Its name appears in the list of asset types in the **Child Asset Field** on the "Add New Association" forms.
7. Click **Save**.

## Step 5: Enable the Asset Type on Your Site

Before you can examine the forms that the new elements render for the asset type, you must enable the asset on the site (or sites) that you are working with and create a simple **Start Menu** item for it.

For instructions on how to enable your asset types and create Start Menu items for them, see the *Content Server Administrator's Guide*.

## Step 6: Fine-Tune the Asset Descriptor File

Create a new asset of your new type and examine the "New," "Edit," "Inspect," and "Search" forms. (To create a new asset of this type, click **New** on the toolbar and select the **Start Menu** shortcut that you created in the preceding procedure. For step-by-step procedures that describe how to create and work with assets, see the *Content Server Advanced Interface User's Guide*.)

After you examine the forms, you might need to modify the asset descriptor file.

You can make any of the following changes with relatively few steps:

- Re-ordering the fields that appear on the CS-Direct forms for the asset type
- Changing the name of a field (that is, the value of `PROPERTY DESCRIPTION`)
- Changing anything in an `INPUTFORM`, `SEARCHFORM`, or `SEARCHRESULTS` statement

If you want to make any of the changes in the preceding list, complete the following steps:

1. Use Content Server Explorer to open and modify the asset descriptor file that you uploaded in "Step 2: Upload the Asset Descriptor File to Content Server" on page 288.
2. Save your changes.
3. Re-register the elements for the asset type.

You cannot change the schema of the asset type's database table after it is created. The following changes are schema changes:

- Changing the name of a column (the `NAME` parameter in an existing `PROPERTY` statement)
- Changing the data type of the column (the `STORAGE TYPE` or `LENGTH` in an existing `PROPERTY` statement)
- Adding a new property (new `PROPERTY` statement), which adds a new column to the table
- Deleting a property (an existing `PROPERTY` statement), which deletes a column from the table

Therefore, if you want to make any of the changes in the preceding list, you must first delete the asset type, modify the asset descriptor file, and then create the asset type again.

Note that if you have customized the elements that AssetMaker copied from the `AssetStubElementCatalog` table to the asset type's directory in the `ElementCatalog` table, your changes are overwritten when you re-register the elements.

## Step 7: (Optional) Customize the Asset Type Elements

There are two ways to customize asset type elements on your content management system:

- If you require changes that are specific to a certain asset type, you can modify the elements for that asset type in the `ElementCatalog` table, using Content Server Explorer. For information about modifying the elements, see "About PreUpdate and PostUpdate Elements" on page 293.
- If you require the same modifications for assets of all types, modify the source elements in the `AssetStubElementCatalog` table, using Content Server Explorer, before you create your asset types. That way you will have to customize only once.

**Stub Element Customization**

Although customizing source elements in the `AssetStubElementCatalog` table might be necessary, it is not supported. If you plan to customize a stub element, consider that it will be overwritten under the following conditions:

- Changing the stub elements requires you to re-register all of the asset elements that must take the new changes.When you re-register asset elements, AssetMaker moves new copies of the elements from the `AssetStubElementCatalog` table to the asset type's directory in the `ElementCatalog` table (the path is `ElementCatalog\OpenMarket\Xcelerate\AssetType`) and in the process overwrites the existing elements. If you customized elements in the `AssetType` directory, they will be overwritten.
- The Content Server upgrade process typically installs new source elements in the `AssetStubElementCatalog` table. Code changes in the stub elements are overwritten when you re-register your asset types after upgrade.

Be meticulous about tracking all of your customizations at all times. That way you can re-create your work if necessary.

## About PreUpdate and PostUpdate Elements

Actions or procedures that can be performed on assets are called **functions**. For example, **New**, **Edit**, and **Delete** are all functions that can be invoked by users of the Dash and Advanced interfaces in order to create, edit, and delete assets. Such functions also call the `PreUpdate` and `PostUpdate` elements.

`PreUpdate` and `PostUpdate` elements are used to contain logic that initiates various operations when the elements are called. The `PreUpdate` element is called when a function is invoked; the `PostUpdate` element is called after Content Server writes asset information to the database. Each element contains a variable whose name and value specify conditions that call the element. The variable is named `updatetype` if the element must be called from the Advanced, InSite, and CS-Desktop interfaces, or the XMLPost utility. The variable is named `dashUpdateType` if the element must be called from the Dash interface.

For example, your content managers are working with the Advanced interface and their system is configured to import batches of articles from a wire service. You can have the `PreUpdate` element set the value for the "Source" field to **wirefeed** and the value for the "Byline" field to **API** just before import occurs if you code these operations in the element and set `updatetype=remotepost` as the condition under which the element will be called.

**Note**

`PreUpdate` and `PostUpdate` elements are always called when a Content Server user invokes the New, Edit, or Delete function in the Advanced and Dash interfaces, when assets are saved in the CS-Desktop and InSite interfaces, and when assets are imported through XMLPost. Whether operations are performed via the `PreUpdate` and `PostUpdate` elements depends on how the elements are coded. By default, they are designed for no action.

`PreUpdate` and `PostUpdate` elements are accessible from Content Server Explorer, in the following path:

```
ElementCatalog\OpenMarket\Xcelerate\AssetType
```

Table 7, on page 294 defines the values of the `updatetype` variable.

**Note**

The `PreUpdate` element is called twice if a user saves a new or edited asset in the Advanced interface:

- The first call occurs before the "New" or "Edit" form is rendered.
- The second calls occurs after the user clicks **Save** in the "New" or "Edit" form, but Content Server has not yet written asset information to the database.

The condition above provides the opportunity to perform operations via `PreUpdate` at one or more points once the New or Edit function is invoked: before the "New" or "Edit" form is rendered, before asset information is written to the database, or both, depending on the value of the element's `updatetype` variable.

**Table 7:** Values of the `updatetype` variable

| `updatetype =` | Description |
|---|---|
| `setformdefaults` | When a user invokes the **New** function in the Advanced interface:<br>• The `PreUpdate` element is called before the "New" form is rendered.<br>• For `PostUpdate`, `setformdefaults` is not a legal value for the `updatetype` variable. |
| `create` | When a user saves a new asset in the Advanced interface:<br>• The `PreUpdate` element is called before the new asset is written to the database.<br>• The `PostUpdate` element is called after the new asset is written to the database. |
| `editfront` | When a user invokes the **Edit** function in the Advanced interface:<br>• The `PreUpdate` element is called before the "Edit" form is rendered.<br>• For `PostUpdate`, `editfront` is not a legal value for the `updatetype` variable. |
| `edit` | When a user saves an edited asset in the Advanced interface:<br>• The `PreUpdate` element is called before the edits are written to the database.<br>• The `PostUpdate` element is called after the edits are written to the database. |
| `delete` | When a user deletes an asset, using in the Advanced interface:<br>• The `PreUpdate` element is called before Content Server deletes the asset.<br>• The `PostUpdate` element is performed after Content Server deletes the asset. |
| `remotepost` | When a user invokes the XMLPost function to import an asset:<br>• The `PreUpdate` element is called before the asset is imported.<br>• The `PostUpdate` element is called after the asset is imported. |
| `MSWord` | When a CS-Desktop user saves an asset created or edited in Microsoft Word:<br>• The `PreUpdate` element is called before the new asset or edits are written to the database.<br>• The `PostUpdate` element is called after the new asset or edits are written to the database. |
| `InSite` | When a user saves an asset in the InSite interface:<br>• The `PreUpdate` element is called before the asset is written to the database.<br>• The `PostUpdate` element is called after the asset is written to the database. |

Table 8 defines the values of the `dashUpdateType` variable.

**Note**

In the Dash interface, the New, Edit, and Delete functions call the `PreUpdate` and `PostUpdate` elements. Unlike the Advanced interface, in the Dash interface the `PreUpdate` element is called only once—when a user clicks **Save** in the "New" or "Edit" form. As a result, the window for performing operations via `PreUpdate` opens once—just before Content Server writes the asset to the database.

**Table 8:** Values of the `dashUpdateType` variable (Dash interface)

| dashUpdateType = | Description |
|---|---|
| `create` | When a user saves a new asset in the Dash interface:<br>• The `PreUpdate` element is called before the new asset is written to the database.<br>• The `PostUpdate` element is called after the new asset is written to the database. |
| `edit` | When a user saves an edited asset in the Dash interface:<br>• The `PreUpdate` element is called before the edits are written to the database.<br>• The `PostUpdate` element is called after the edits are written to the database. |
| `delete` | When a user deletes an asset in the Dash interface:<br>• The `PreUpdate` element is called before Content Server deletes the asset.<br>• The `PostUpdate` element is called after Content Server deletes the asset. |

## Step 8: (Optional) Configure Subtypes

For the basic asset types that you design with AssetMaker, subtype means a subclass of the asset type based on how that asset is rendered. You can use subtypes to define a separate default approval template for an asset of that type and subtype on each publishing target.

### Adding Subtypes

#### To create a subtype

1. On the **Admin** tab, expand the **Asset Types** option.
2. Under the **Asset Types** option, select the asset type that you want to create subtypes for.
3. Select the **Subtypes** option.

The “Subtypes” form appears:

4. Click **Add New Subtype**.
5. In the next form, click in the first field in the **Name** column and enter the name of the subtype.
6. In the corresponding field in the **Sites** column, select the names of the sites that need this subtype.
7. Repeat steps 5 and 6 for up to five subtypes.
8. Click **Save**.

## Deleting Subtypes

### To delete a subtype

1. On the **Admin** tab, expand the **Asset Types** option.
2. Under the **Asset Types** option, select the asset type that you want to create subtypes for.
3. Select the **Subtypes** option.
4. In the “Subtypes” form, click the **Delete** (trash can) icon.
5. Click **Delete Subtype.**

# Step 9: (Optional) Configure Association Fields

Named associations are described in “The Basic Asset Model” on page 186. Briefly, named associations are defined, asset-type-specific relationships that describe parent-child relationships or links between individual assets or asset sub-types. When you code your template elements, you use the names of these relationships to identify individual assets or sub-types, without having to refer to the object by its name.

For example, the Burlington Financial sample site associates imagefile assets with article assets. The template elements are coded to extract an associated imagefile asset by the name of the association rather than the name of the asset. That way, if you choose a different imagefile for an article, the template displays the new imagefile without your having to recode the template.

Named associations are represented as fields in the asset forms. These fields are not created from an asset descriptor file. Instead, you use the “Asset Associations” forms on the **Admin** tab.

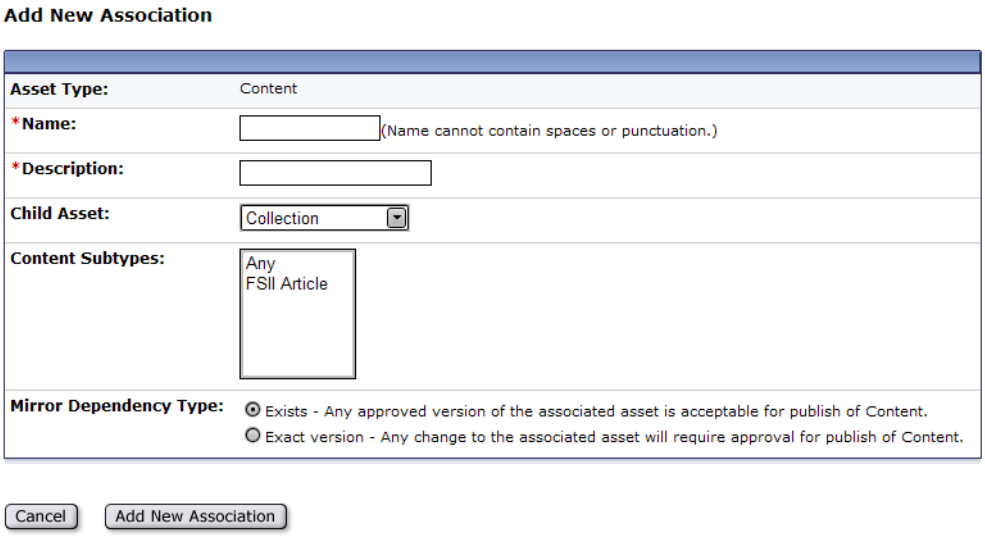
Examples of association fields from the Burlington Financial sample site include the Main Image and Teaser Image associations between article assets and imagefile assets. When you create a Burlington Financial article asset, you can select an image asset from the **Main Image** and **Teaser Image** fields.

## Adding Association Fields

### To add an association field

1. On the **Admin** tab, select **Asset Types.**
2. Under the **Asset Types** option, select the asset type that you want to create associations for.
3. Under the asset type you selected, select **Asset Associations > Add New.**

   The "Add New Association" form appears:




4. Click in the **Name** field and enter the name of the association field, without using spaces, decimal points, or punctuation marks.
5. Click in the **Description** field and enter a short description of the field. Keep the description short because CS-Direct uses the text entered into this field as the name of the field when it is displayed on the new asset form.
6. Click in the **Child Asset** field and select the kind of asset type that will appear in this field. (It is called the **Child Asset** field because associations create parent-child relationships between assets.) You cannot specify the template or the page asset type in this field.
7. Select the subtype or subtypes for this association by highlighting them in the **Subtypes** field. To select multiple subtypes, press the **Control** key while you click your selection with the mouse.
8. Select the appropriate **Dependency Type** for this asset association. By default, it is set to **Exists**. (The dependency type specified here is used by the approval system when your publishing method is Mirror to Server. For information about dependency types, see "About Coding to Log Dependencies" on page 585).
9. Click **Add**.

   CS-Direct creates a row in the `Association` table for this association. The name used in the row is the text you entered in the **Name** field in step 4.

## Deleting Association Fields

Before you delete an association field, be sure to search for any assets that use it and clear the value in the field. Otherwise, those assets will still have the association when you delete the association field, but, because the field is no longer displayed in the Content Server interface, you will be unable to change it.

### To delete an association field

1. On the **Admin** tab, select **Asset Types.**
2. Under the **Asset Types** option, select the asset type whose association field you want to delete.
3. Under the asset type you selected, select the association that you want to delete.
4. In the form on the right, click **Delete**.

The association field is now no longer displayed on forms for this asset type.

# Step 10: (Optional) Configure Categories

You can use categories to organize your asset types according to some convention that works for your site design. For example, the Burlington Financial sample site uses queries based on category to determine which articles should be selected for various sections of the online site.

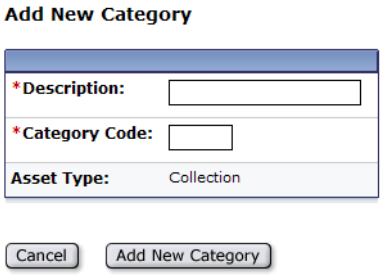
Although all basic asset types have a **Category** field (column) by default, it is not a required field and you do not have to use it.

## Adding Categories

### To add a category

1. On the **Admin** tab, select **Asset Types.**
2. Under the **Asset Types** option, select the asset type that you want to create categories for. (For example, select **Article**.)
3. Under that asset type, select **Categories > Add New**.

   The following form appears:




4. Click in the **Description** field and enter a short description of the category. Keep the description short because CS-Direct uses the text that you enter in this field in the site tree and in the drop-down list for the **Category** field on the forms for assets of this type.
5. In the **Category Code** field, enter a two-character code for your new category.

6. Click the **Add** button.
7. Repeat steps 4 through 8, as needed, to finish creating the categories for this asset type.

The categories you created now appear in the drop-down lists in the **Category** fields on the "New" and "Edit" asset forms.

## Deleting Categories

### To delete a category

1. On the **Admin** tab, select **Asset Types**.
2. Under the **Asset Types** option, select the asset type that you want to delete a category for.
3. Under that asset type, select the category that you want to delete.
4. Click **Delete**.

# Step 11: (Optional) Configure Sources

Sources apply to all the asset types in all the sites on your system. Therefore, if you are using Source, you need to add your sources only once (not for each asset type).

## Adding Sources

### To create a source

1. On the **Admin** tab, select **Sources** > **Add New**.

   The "Add Source" form appears:

2. Click in the **Source** field and enter the name of a source.
3. Click in the **Description** field and enter a short description of the source. Keep this description short because CS-Direct uses the text from this field in the drop-down list for the **Source** field on the forms for assets.
4. Click **Add**.

   The source is written to the `Source` table.
5. Repeat steps 1 through 4 for each source that you need for your asset types.

## Deleting Sources

### To delete a source

1. On the “Admin” form, select **Sources**.
2. Under the **Sources** option, select the name of the source that you want to delete.
3. In the form for this source, click **Delete**.

## Step 12: (Conditional) Add Mimetypes

The `MimeType` table holds mimetype codes that can be displayed in mimetype fields. You must add mimetypes for your asset if you reference the `MimeType` table in your asset descriptor file.

For example, both the imagefile and stylesheet asset types have upload fields with **Browse** buttons next to them. After you select a file in the upload field, you specify the mimetype of the file you selected from the **Mimetype** field.

The **Mimetype** fields for the imagefile and stylesheet asset types query the MimeType table for mimetype codes based on the `keyword` column:

- Mimetype codes with their `keyword` set to `stylesheet` appear in the drop-down list of the **Mimetype** field in the “Stylesheet” form.
- Mimetype codes with their `keyword` set to `image` appear in the drop-down list of the **Mimetype** field in the “ImageFile” form.

By default, stylesheet files can be `CSS` files and imagefile files can be `GIF` or `JPEG` files. You can add mimetype codes for these asset types, for your own custom asset types, or for any other reason.

#### To add mime types to a Mimetype drop-down list

1. Open Content Server Explorer.
2. Expand the **MimeType** table.
3. Do one of the following:
   - If you are adding a mimetype for the imagefile asset type, select the **image** folder in the **MimeType** table.
   - If you are adding a mimetype for the stylesheet asset type, select the **text** folder in the **MimeType** table
   - If you are adding a mimetype for a custom asset type with an upload field or for any other reason, select the appropriate location in the **MimeType** table.
4. Right-click in the frame on the right and then select **New** from the drop-down list.

   Content Server Explorer creates a new row in the table.
5. In the **mimetype** field, enter the name of the mimetype. For example: `XSL`.
6. In the **extension** field, enter the extension for mime types of this type. For example: `.xml`.
7. In the **description** field, enter a short description of this mimetype.
8. In the **isdefault** field, do one of the following:
   - If you want to specify more than one extension for the same mimetype, enter `n`. For example, if a mimetype named JPG has `.jpg` and `.jpeg` extensions, set **isdefault** to `n`.
   - If this is the only extension for the mimetype, enter `y`.

9. Click in the **keyword** field and do one of the following:
   - If you are adding a mimetype for the imagefile asset type, enter `image`.
   - If you are adding a mimetype for the stylesheet asset type, enter `stylesheet`.
   - If you are adding a mimetype for a custom asset type with an upload field or for any other reason, enter the appropriate keyword.
10. Select **File > Save all**.

    Content Server Explorer saves the row.

    If you added a mimetype code with the keyword of `image`, that mimetype is now displayed in the **Mimetype** field of the "ImageFile" form. If you added a mimetype code with the keyword of stylesheet, that mimetype is now displayed in the **Mimetype** field of the "Stylesheet" form.

## Step 13: (Optional) Edit Search Elements to Enable Indexed Search

CS-Direct (and CS-Direct Advantage and Engage) has its own database SQL search mechanism that runs the Simple and Advanced searches. However, you can set up your management system to one of the supported third-party search engines instead. (See the *Content Server Administrator's Guide* for configuration information.)

When you are using a search engine on your management system, each asset is indexed when it is saved after being created or edited. By default, only the default fields are indexed (for a list, see "Default Columns in the Basic Asset Type Database Table" on page 191). If you want the fields that you created with `PROPERTY` statements in your asset descriptor file to be indexed, you must add statements for them in the following elements:

- `OpenMarketXcelerate/AssetType/`*`YourAssetType`*`/IndexAdd.xml`
- `OpenMarketXcelerate/AssetType/`*`YourAssetType`*`/IndexReplace.xml`

To add the asset type's custom fields to these elements, use the Content Server `INDEX` tags and follow the convention illustrated in these elements.

## Step 14: Create and Assign Asset Type Icons (Dash Interface only)

When a user performs a search or runs a tag in the Dash interface, the list of returned assets displays, for each asset, an icon representing the asset's type in the "Type" column. (Hovering the mouse cursor over the icon will display the asset type.)

You must create and assign an icon that uniquely identifies your new asset type by completing the steps below.

1. Create a GIF image no larger than 20x20 pixels representing the asset type.
2. Name the file using the syntax *`assetType`*`.gif`. The file name determines the asset type for which the icon will be displayed. The name is case-sensitive.
3. Place the file in the following directory:

   `<cs_app_dir>/Xcelerate/OMTree/TreeImages/AssetTypes/`

   where `<cs_app_dir>` is the directory of the deployed CS application on your application server.
4. Restart your application server for the icons to appear in the Dash interface.

## Step 15: Code Templates for the Asset Type

Creating your asset types and coding the templates for assets of that types is an iterative process. Although you need to create asset types before you can create templates for assets of that type, it is likely that you will discover areas that need refinement in your data design only after you have coded a template and tested the code.

For information about coding templates, see Chapter 27, "Coding Elements for Templates and CSElements."

## Step 16: Move the Asset Types to Other Systems

When you have finished creating all of your new asset types (including creating templates for them), migrate them to the management and delivery systems. System administrators will then configure the asset types for the management system. They will enable revision tracking where appropriate, create workflow processes, create Start Menu shortcuts, and so on.

For information about this step, see the *Content Server Administrator's Guide*.

# Deleting Basic Asset Types

When you delete an asset type that you created with AssetMaker, CS-Direct can either delete components of the asset type that you select, or delete all traces of the asset type from the database. A list of asset type components follows:

- The database table and **all the data in it**
- The elements in the `ElementCatalog` table
- The SQL statements in the `SystemSQL` table
- The row in the `AssetType` table
- Any rows in the `Association` table (optional)
- Any rows in the `Category` table (optional)
- Any rows in the `AssetPublication` table
- Any rows in the `AssetRelationTree` table

### To delete an asset type that you created with AssetMaker

1. Open your browser and enter this address:

   `http://`*`your_server`*`/Xcelerate/LoginPage.html`

2. Enter your login name and password and click **Login**. Note that you must have administrator rights associated with your user name (login name) in order to have access to AssetMaker.

3. Select the **Admin** tab > **AssetMaker** and then select the asset type that you want to delete.

   The "Asset Type" form appears:

4. Select the **Delete Asset** option and click **Submit**.

   CS-Direct displays a confirmation message.

5. Click **OK**.

# Images and eWebEditPro

The eWebEditPro toolbar allows users to upload an image to the management system. However, if your delivery system is dynamic, think carefully before allowing your users to take advantage of this feature. An image that is uploaded in this manner is not an asset, and so it is **not** mirror published when the asset that uploaded it is published to the delivery system. You must set up a separate file-transfer process to ensure that those images are moved to the delivery system. To avoid the extra step, consider making your images assets and using the association fields to connect an image to another asset.

Chapter 16

# Designing Flex Asset Types

As mentioned in Chapter 11, "Data Design: The Asset Models," the data model for flex asset types can be thought of in terms of a family of asset types, with each asset type in the family having several database tables.

You create new flex asset types with the Flex Family Maker utility. However, when working with the flex asset model, developers not only create the flex asset types, they also create the individual data structure assets of those types—that is, flex attributes, flex parent definitions, flex definitions, and flex parent assets.

Typically, you design the flex asset types and create the data structure assets on a development system. Then when your data model is ready, you migrate your work from the development system to the management and delivery systems.

This chapter contains the following sections:


- Design Tips for Flex Families
- The Flex Family Maker Utility
- Creating a Flex Asset Family
- Editing Flex Attributes, Parents, and Definitions
- Using Product Sets

# Design Tips for Flex Families

Your job when designing your flex family is to create a data structure that meets the needs of two audiences:

- The visitors to your online site (that is, the users of your delivery system)
- The content providers who use CS-Direct Advantage to enter data into the Content Server database (that is, the users of the management system)

## Visitors on the Delivery System

The experience of the visitors to your online site is based on the following asset types:

- flex asset
- flex attribute

Your online site pages display flex assets (assetsets) for the visitors through queries that are based on attribute values (searchstates. See "Assetsets and Searchstates" on page 206.) You use attribute values as the basis for drill-down searches that can give the appearance of a hierarchy on your online site if that is the look and feel that you want.

## Users on the Management System

The users of your management system navigate through a visual hierarchical structure that you create for them with the following flex asset types:

- flex parent definition
- flex definition
- flex parent

Although the organizational structure that you create with these asset types does affect the data — it determines which attribute values are inherited by which flex assets—its biggest impact is on the users of the management system.

You are not required to use flex parents and flex parent definitions, but their inheritance properties make them a valuable tool for users who are maintaining a large amount of data such as an online catalog:

- Changing an attribute value at the parent level changes that value for all the flex assets who are children of that parent, which means you only have to change the value once.
- Inherited attribute values are values that aren't subject to user error during data entry, which means less data cleanup is required.

The inheritance tree that you create for your content providers has no bearing on how your site visitors navigate the online site you are designing. For example, if content is entered into your management system through some completely automated process—perhaps it is bulk loaded from an ERP system—you would have no need for parent asset types at all, yet you can still create drill-down searches on your online site.

## How Many Attribute Types Should You Create?

As described in "Assetsets and Searchstates" on page 206, only the flex assets that share a common attribute type can belong to the same assetset because queries (searchstates) are based on attributes and not on the organizational constructs of parent definitions and flex definitions.

Therefore, when you design your data structure, remember that if you organize your data to use separate types of attributes, you might create a nicely delineated interface on the management system, but that data cannot be synthesized well on the delivery system and that is rarely what you want.

As a general rule, you should **create one type of attribute** for your system. If you need to, you can create more than one version of the rest of the family members (the flex asset type, flex definition type, flex parent type, and flex parent definition type), but they should still share the same pool of attributes.

For example, if the GE Lighting sample site had been designed such that the product family and the content family shared the same attribute type, you would be able to create assetsets that contained a product and a corresponding article about that product.

## Designing Flex Attributes

Before you begin creating attributes, design them on paper. Determine all the attributes you need and decide where they will appear—with flex assets or the flex parents.

Start by planning out the bottom level of your hierarchy (that is, the individual instances of flex asset types like products) and determine the attributes you need for each item at that level. For example, if you plan to create flex filter assets, determine which attributes need to be created and assigned to the definitions as the input and output attributes for your filters.

You must determine all of the flex attributes that you need before you begin creating them because the way you plan to use them creates dependencies that you must account for when you create them.

### Which Data Types

Assess the data types that are available for attributes and the default input types for those data types. Determine which data types will work best for which attributes. If you want to change the default input style for an attribute, you create an attribute editor for it before you create the attribute. (See Chapter 18, "Designing Attribute Editors.")

When you create a flex asset that uses an attribute of type `blob`, the format of the value entered for the attribute on an **Inspect** form depends on its type. For example, a text file shows the first 200 bytes in the file. An image file appears as a thumbnail image. And some files cannot be displayed at all. In this case, CS-Direct Advantage displays the message "***filename* not displayable**" but the file location is still successfully recorded.

### Using Attribute Editors

The default input type for an attribute depends on the data type that you select for it. If you do not want to use the default input type, you can create an attribute editor for the attribute.

Creating flex assets and their attribute editors is an iterative process. You can create the attribute editors first or you can create the attributes first and then go back and assign the

attribute editors after you have created them. The process of creating attribute editors is described in Chapter 18, "Designing Attribute Editors."

### Attributes of Type Blob

The default input style of an attribute of type `blob` is a text field with a **Browse** button. You use the **Browse** button to locate and select a file and Content Server uploads it to the default storage directory. You cannot use the CS-Direct Advantage forms to edit the contents of the file.

If you want to be able to enter content directly into the external file through the CS-Direct Advantage forms, you must assign an attribute editor to the attribute:

- If you use an attribute editor that uses the `TEXTAREA` input style, you can create a field that can hold up to 2,000 characters (entered through the forms); when saved, that content is written to the default storage directory.
- If you have eWebEditPro, you can use an eWebEditPro field to edit the contents of the external file that the attribute represents.

### Attributes of Type Asset

The default input style for an attribute of type asset is a pull-down list of all the assets of the type specified. An unfiltered pull-down list is not recommended if you have more than 20 assets of that type.

In general, whenever you create an attribute of type asset, you should assign it an attribute editor.

- An attribute editor that uses the `PICKASSET` style checks to find out whether the tree is toggled on or off in the Content Server interface. If the tree is on, the user can select an asset from a tab in the tree. If the Tree is toggled off, the attribute editor displays a pop-up window that lists the assets from the **Active List** and `History` tabs.
- Another option is to use the `PULLDOWN` style but to supply a query asset that limits the options that appear in the list.
- If the number of assets that are valid choices is small, you can also use the `CHECKBOXES` or the `RADIOBUTTONS` input style, both of which require a query asset to identify the assets.

## Where Will Each Attribute Be Used?

After you have determined the list of attributes, determine whether you plan to use them in a flex definition or a flex parent definition. Sort them logically by using the following guidelines:

- If an attribute's value is unique to an individual flex asset (product, article, image, for example), the attribute belongs at the bottom of the tree, with the flex asset.
- If an attribute's value is the same for multiple flex assets, the attribute belongs in a parent. (Of course there are always exceptions. For example, even if a toaster costs the same amount as a bowling ball, it is unlikely that they would inherit their prices from a common parent.)
- Based on that attribute distribution, you can determine how many flex definitions you need and how many parent definitions you need.

Remember that there is both a physical limit (based on your DBMS) and a psychological limit (user satisfaction) as to how many attributes you can or should use in an individual flex asset or flex parent. Someone has to enter all those values. Be sure to create and then

assign to the definitions only those attributes that you really plan to use. It is very easy to add attributes in the future if you decide that you need additional ones.

### Dependencies Imposed by Hierarchy

After you know where an attribute will be used, you can determine whether hierarchical concerns add requirements to the attribute. For example, if an attribute is to used by a flex parent and your data structure allows flex assets to have more than one parent, the attribute must be configured to hold multiple values because a flex asset might inherit more than one value for it.

In general, try not to make the inheritance structure too complex.

## How Many Definition Types Should You Create?

The appearance and input of data on the management system is based on the flex asset definitions and the flex parent definitions. Parents and flex assets appear on tabs in the tree in the Content Server interface based on the hierarchy that you create through the definitions.

In general, it is best to create a separate set of definition types for each flex asset member in a family.

For example, the GE Lighting sample site has two flex asset members in the content family: article (flex) and image (flex). They share parents, parent definitions, and flex definitions. This means that some attributes are left blank for the image assets because they don't apply and some attributes are left blank for the article assets because they don't apply.

It would be better to have article parents, article definitions, and article parent definitions that are different from image parents, image definitions, and image parent definitions. But they should absolutely share the same attribute type, which they do.

## Designing Parent Definition and Flex Definition Assets

The hierarchy on the tabs in the tree in the Content Server interface is created through the flex parent definitions and flex definitions:

- To set a hierarchy three levels deep, you need at least two parent definitions and at least one flex definition.
- To specify a hierarchy two levels deep, you need at least one parent definition and at least one flex definition.

Be sure to consider the basic tenets of usability when you set up a structural hierarchy with the flex definitions and flex parent definitions. For example:

- How deep can the hierarchy go before the content providers feel lost in the tree?
- How many attribute values can be inherited to alleviate the possibility of user error during input?
- How many options can be comfortably displayed in a drop-down list?

If you create a system that is overly difficult to use, the content providers will complain.

### Determining Hierarchical Place

Open CS-Direct Advantage, log in to the GE Lighting sample site, and examine the form for a new product parent definition or for a new product definition.

In the **Parent Definition** section of these forms, you determine two things:

- The hierarchical position of the assets that use this definition and determine
- The parents that they can inherit attributes from

Remember that although the hierarchical position has meaning only in the user interface on the management system, the attributes that they inherit have meaning both on the management system and on your online site.

The text box named **Available** lists all the existing parent definitions. You use this section of the form to specify how many parents are possible by selecting parent definitions from the **Available** list and moving it to the **Selected** list.

When you create a parent asset or a flex asset, the **New** form displays a drop-down field for each definition that you selected from the **Available** list when you created the definition that you are using to create the new parent or flex asset. The drop-down list in the **New** form displays all the parents that were created with that definition.

If the parent that is selected in the **New** form has any attribute values, the asset inherits them.

How many possible parents should you allow? In general, it is best to keep this simple. The more parent definitions you select from the **Available** list, the more fields the content providers have to fill out when they create a new flex asset.

If you do not select a parent definition in the **Available** list, it means that assets created with this definition are positioned at the top level of the tree on the tab that displays your flex assets.

The best way to understand how parent definitions, flex definitions, parent assets, and flex assets interact is to examine the assets delivered with the GE Lighting sample site.

## Determining Attribute Inheritance

You configure attribute inheritance in the **Attributes** section of the parent definition form. You use that section to specify the attributes that define the parents that are created with this definition.

When you create a parent with this definition, the values that are entered for these attributes are passed down to the flex assets that are children of the parent asset.

## How Many Flex Parent Definition Assets?

The simple answer is “as many as you need.” Be sure to consider usability when you decide how many flex parent definition assets you need, and how many parent assets of those definitions that you need.

If you create many parent definitions, it probably means that you will have fewer parents created with each definition, which leads to shorter drop-down lists in the new parent and new flex asset forms. Short drop-down lists make it easier for content providers to select the correct parent from the list.

However, if your data needs require you to have a small number of parent definitions and a large number of parents, create a tab that lists all the parents so the content providers can select the correct parent asset from the tab.

## How Many Flex Definition Assets?

A general rule is this: create enough flex definitions so that fields (attributes) are not left blank on the **New** and **Edit** flex asset forms.

If you create too few definitions, you run the risk of creating long forms with lots of attribute fields, not all of which apply for each asset. When you have long forms with lots of attribute fields, not only do content providers have to sort through the form to determine which attributes apply to the asset they are currently creating, the form takes a long time to be rendered in the user's browser.

## Summary

Keep the following rules in mind as you design the data structure with a flex family for your online site:

- Carefully planned, easy-to-use asset design (data design) makes content providers happy.
- Usable layout and efficient code makes site visitors happy.

And both user groups need efficient systems that perform well.

# The Flex Family Maker Utility

When you create a flex family with Flex Family Maker, it does the following:

- Creates several database tables (the number depends on which flex asset types that you create).
- Writes information about the new flex family to the following tables:
  - `FlexAssetTypes`, which holds a row for each flex asset member type
  - `FlexGrpTmplTypes`, which holds a row for each flex parent definition type
  - `FlexGrpTypes`, which holds a row for each flex parent type
  - `FlexTmplTypes`, which holds a row for each flex definition type
- Creates new directories in the `ElementCatalog` table using the following naming convention:

  `OpenMarket/Xcelerate/AssetType/`*`NameOfYourAssetType`*
- Copies elements from the `ElementCatalog` table to the directories in created for your asset types. CS-Direct Advantage use these elements to format the **New**, **Edit, Inspect**, **Search**, and **Search Results** forms for assets of that type.

For information about the main database tables for flex assets and flex parent assets, see "Flex Families and the Database" on page 203. For information about all the database tables in a flex family, see the *Content Server Database Schema Guide*.

## The Flex Asset Elements

When you create a new flex asset type, Flex Family Maker copies elements to the following location in the `ElementCatalog` table:

`OpenMarket/Xcelerate/AssetType/`*`NameOfAssetType`*

For example, the GE sample site product asset elements are in:

`OpenMarket/Xcelerate/AssetType/Products`

It also creates a SQL statement that the search elements use and places it in the `SystemSQL` table under `OpenMarket/Xcelerate/AssetType/`*`NameOfAssetType`*.

For a description of the elements and the SQL statement that Flex Family Maker copies for you, see "Elements and SQL Statements for the Asset Type" on page 274. The elements for flex assets are the same as the elements for the basic assets with the exception of the `AppendSelectDetailsSE` element.

# Creating a Flex Asset Family

When you are using the flex asset data model to represent the content you want to display on your online site, you and the other developers do not create only the flex asset types. You also create the individual data structure assets of those types: flex attributes, flex parent definitions, flex definitions, and flex parent assets.

## Overview

Following is an overview of the process for creating a new flex asset type or family of flex asset types. Where you start in the process depends how many asset types you need to design. If you can base your data structure on either of the sample site flex families, you do not have to create an entire flex family—you can create only the new members that you need.

This chapter describes each of the following steps in the process, except as noted:

1. Create the new flex family (page 313).
2. Create additional flex family members, as necessary (page 315).
3. Enable the new asset types on all the Content Server sites on the development system (page 316) and create **Start Menu** shortcuts for all the new asset types. Also, place the new flex definition, flex parent definition, and attribute types on the **Design** tab, and create a tab for your new flex parent and flex asset types.
4. Create the flex attributes and design your attribute editors (page 317). For more information about attribute editors, see Chapter 18, "Designing Attribute Editors."
5. (Optional) Create the flex filter assets (page 321). (For more information about flex filters, see Chapter 17, "Flex Filters.")
6. Create the flex parent definitions (page 323).
7. Create the flex definition assets (page 324).
8. Create the flex parent assets (page 326).
9. Create and assign icons that will represent the members of your flex family in the Dash interface (page 328). The icons will be displayed in the "Type" column whenever a user performs a search or runs a tag.
10. Create templates for the flex assets, the flex member of the flex family (page 328).
11. Test your design by creating enough flex assets to examine the data structure that you have designed (page 328). (Procedures for creating assets are presented in the *Content Server Advanced Interface User's Guide*.)
12. (Optional). Create flex asset associations (page 328).
13. Move your asset types to other systems (management and delivery; page 329). This step is described in the *Content Server Administrator's Guide*.

## Before You Begin

Set up your development system and get access to it, as follows:

- Create the appropriate Content Server sites.
- Create a user name for yourself that has administrator rights, and enable that user name on all of the sites on your development system. Note that without administrator rights, you do not have access to the **Admin** tab, which means that you cannot perform some of the procedures in this chapter.

  For the sake of convenience, assign the **Designer** and **GeneralAdmin** roles to your user name. That way you will have access to all the tabs in the Content Server interface and all of the existing **Start Menu** shortcuts for the assets in the sample site. (Be sure that the TableEditor ACL is assigned to your user name.)
- If you plan to use eWebEditPro, a third-party HTML editor from Ektron, Inc., you must configure it on the systems that you plan to use it on. It is not delivered with CS-Direct (or CS-Direct Advantage).

For information about these tasks, see the *Content Server Administrator's Guide*.

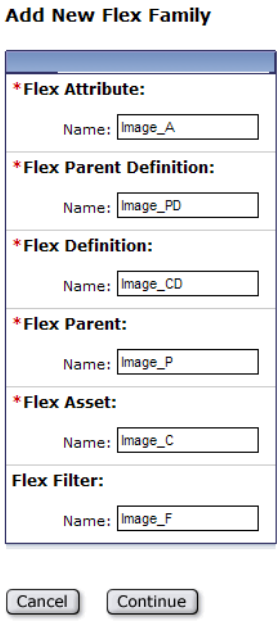
## Step 1: Create a Flex Family

### To create a new flex family

1. Open your browser and enter this address:

   `http://your_server/Xcelerate/LoginPage.html`
2. Enter your login name and password and click **Login**. Note that you must have administrator rights associated with your user name (login name) in order to have access to the **Admin** tab, which is where **Flex Family Maker** is located.
3. Select **Admin > Flex Family Maker**.

   The "Add New Flex Family" form appears:



4. For **Flex Attribute**, click in the **Name** field and enter the name of the new flex attribute asset type.

   The name you enter in this field is the internal name of the new attribute asset type. It becomes the name of the core table for this asset type and the prefix for all its auxiliary tables.

5. For **Flex Parent Definition,** click in the **Name** field and enter the name of the new flex parent definition asset type.

   The name you enter in this field is the internal name of the new parent definition asset type. It becomes the name of the core table for this asset type and the prefix for all its auxiliary tables.

6. For **Flex Definition**, click in the **Name** field and enter the name of the new flex definition asset type.

   The name you enter in this field is the internal name of the new flex definition asset type. It becomes the name of the core table for this asset type and the prefix for all its auxiliary tables.

7. For **Flex Parent**, click in the **Name** field and enter the name of the new flex parent asset type.

   The name you enter in this field is the internal name of the new parent asset type. It becomes the name of the core table for this asset type and the prefix for all its auxiliary tables.

8. For **Flex Asset,** click in the **Name** field and enter the name of the new flex asset type.

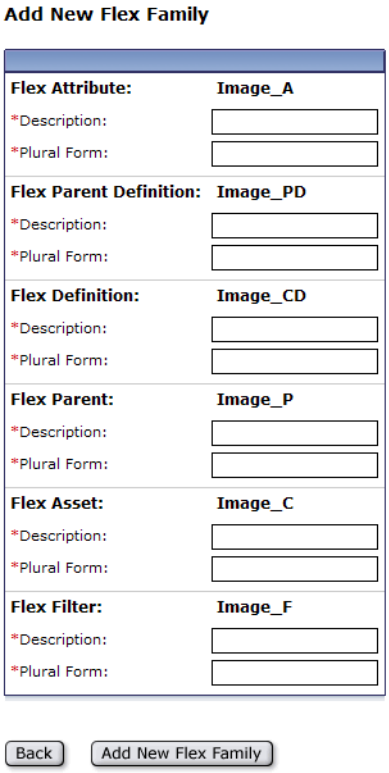
   The name you enter in this field is the internal name of the new flex asset type. It becomes the name of the core table for this asset type and the prefix for all its auxiliary tables.

9. For **Flex Filter**, click in the **Name** field and enter the name of the new flex filter asset type.

   The name you enter in this field is the internal name of the new flex filter asset type. It becomes the name of the core table for this asset type and the prefix for all its auxiliary tables.

10. Click **Continue**.

Flex Family Maker displays the following form:




11. For each new member of the family, click in the **Description** field and enter the external name of the asset type, that is, the name of the asset type when it is displayed in CS-Direct Advantage. This is the name that appears on the forms (**New**, **Edit**, **Inspect**, and so on).
12. For each new member of the family, click in the **Plural** field and enter the plural version of its name. This version is used in status messages and so on when appropriate.
13. Click **Add New Flex Family**.

    Flex Family Maker creates the database tables that will store assets of these types. For information about these tables, see "Flex Families and the Database" on page 203.

    It also copies elements that format the forms for assets of these types to a directory with the name of the asset type in the `ElementCatalog` and `SystemSQL` tables.

## Step 2: (Optional) Create Additional Flex Family Members

If you need to create additional flex family members (for example, if you need more than one flex asset type per member category), do the following:

1. In the **Admin** tab, expand the flex family you just created.
2. Drill down the flex family tree until you reach the category (flex parent for example) for which you want to create another member.

3. Under the desired member category, double-click **Add New *Member Category* Asset Type.**
4. Content Server displays the "New *Member Category* Asset Type" form.
5. In the form, fill out the required fields and click **Save**.

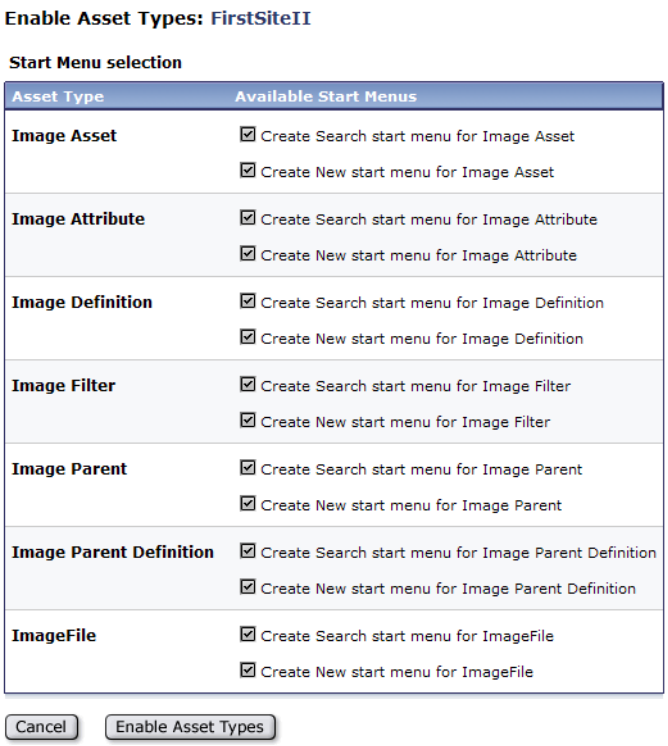
   Content Server displays a message confirming that the asset type was created.
6. Repeat this procedure for each additional flex family member you want to create.

## Step 3: Enable the New Flex Asset Types

Before you can start creating assets (attributes, flex parent definitions, and so on), you must complete some steps on the **Admin** tab so that you have access to them. Note that your login must grant you administrator rights in order for you to have access to the **Admin** tab.

Complete the following steps:

1. On the **Admin** tab, click the **Sites** icon and complete the following steps:
   a. Select the site that you are going to use to work with this asset type.
   b. Under that site, select **Asset Type** > **Enable Asset Types**.
   c. Select your new asset types from the list and click **Enable Asset Types**.
   d. Content Server can automatically create a **New Start Menu Item** and/or a **Search Start Menu Item** for the Asset Types you are enabling. Check the box next to any available Start Menu Item that you would like Content Server to create.



If you choose not to generate these menu items at this time, you or your site administrator must manually create them later (no one can create assets of the enabled asset types until Start Menu items are created for them).

e. Repeat steps a through d for each appropriate site.

2. Click **Enable Asset Types**.

3. The asset types are now enabled for the site(s). If you did not use Content Server to generate start menu items, you or your site administrator must now manually create them. As the developer of the asset types and the designer of the online site, your responsibility is to let the administrator know enough about your asset and site design that the site administrator can configure meaningful Start Menu items.

   You (the developers) must let the site and system administrators know which fields are used by the queries, collections, or other design elements for your online site so that they can create meaningful Start Menu items for the content providers. For more information about creating Start Menu items, see the *Content Server Administrator's Guide*.

After you or your administrator has created Start Menu items for your new asset types, you can create assets of these types. Note that even if you add your asset types to a tab, you will not be able to create new assets until you have created Start Menu shortcuts for them.

## Step 4: Create Flex Attributes

Because the steps that you follow can differ significantly based on the data type that you select for your attribute, this section presents several procedures:

- A basic procedure for creating attributes of most data types
- Creating attributes of type `blob`

**Note**

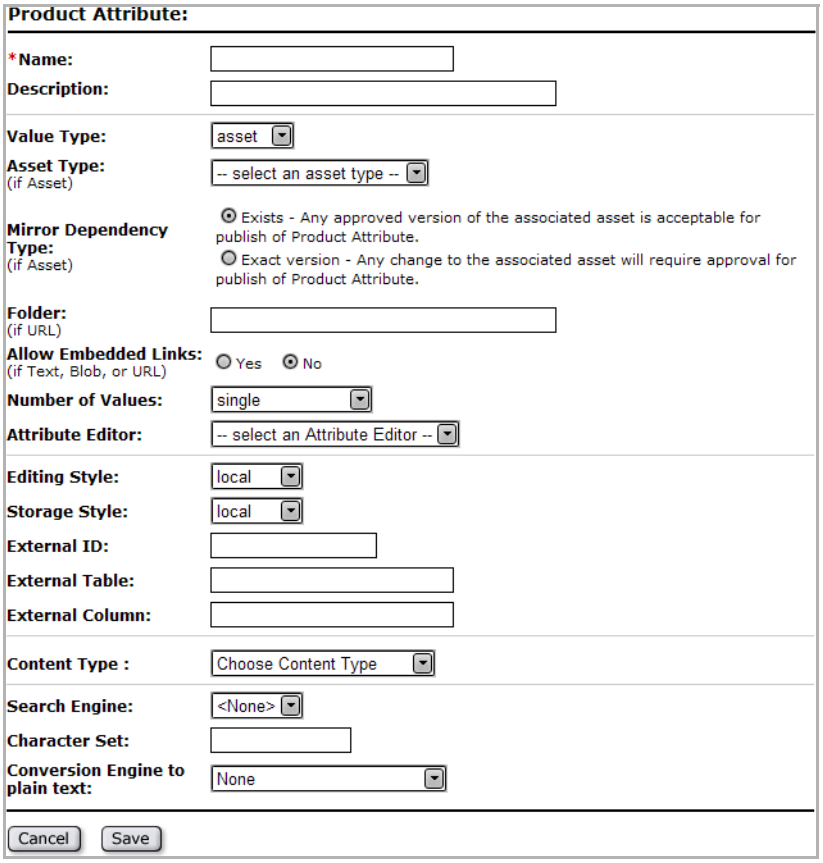
The `url` data type has been deprecated in the 4.0 version of the product. Use the `blob` data type instead.

- Creating attributes of type `asset`
- Creating foreign attributes (that is, attributes that are stored in a foreign table). You must register the foreign table that contains the data you wish to use as a flex attribute. For information, see "Registering a Foreign Table" on page 225.

## Creating Flex Attributes: Basic Procedure

1. If the Content Server interface is not open, log in and select the appropriate site.
2. Click **New** and select the name of your attribute type from the list of shortcuts.

   The **New** attribute form appears. For example, here's the **Product Attribute** form from the GE Sample site:




3. Click in the **Name** field and enter a name of up to 64 characters, excluding spaces.
4. Click in the **Description** field and enter a short, descriptive phrase that describes the use or function of the attribute.
5. Click in the **Value Type** field and select a data type for this attribute.
   - If you select **blob**, see "Creating Flex Attributes of Type Blob (Upload Field)" on page 319 and start with step 3 on page 319.
   - If you select **asset**, see "Creating Flex Attributes of Type Asset" on page 320.
   - If you wish to create a foreign attribute, select a value other than **asset** or **blob**, complete step 6 on page 319, and continue with step 3 on page 321.
   - If you select **text**: The Content Server Advanced interface supports searches on attribute values of all types except **text**. For information about performing searches on attribute values, see the *Content Server Advanced User's Guide*.

   If you need help deciding which data type is appropriate for your attribute, see "Data Types for Attributes" on page 199.

6. Click in the **Number of Values** field and select either **single** or **multiple** from the drop-down list, as appropriate for the data type that you selected in the **Value Type** field.

   If this attribute is to be used by a flex parent and your data structure allows multiple flex parents for a flex asset, you must select **multiple** because the flex assets that inherit values for this attribute might inherit a value from more than one parent.

   **Note**

   When an attribute is configured to accept multiple values, it appears on the flex parent and flex asset forms as a field with an **Add Another *attribute name*** button.

   If you want the attribute to accept multiple values for inheritance reasons but you do not want content providers to select more than one value for the attribute for individual parents or flex assets, assign the attribute an attribute editor that presents it as a single-value field (but select multiple in the **Value Type** field).

   Content Server does not allow attributes of type `text` to have multiple values, due to the way these attributes are stored in the database. A pop-up message denoting this restriction appears if you attempt to save an attribute configured in such a way.

7. (Optional) If you do not want to use the default input type for this attribute (which is based on the data type that you selected in the **Value Type** field), click in the **Attribute Editor field** and select one from the drop-down list.

   If you need more information:

   For a list of the default input types (so you can determine whether you want to use an attribute editor instead), see "Default Input Styles for Attributes" on page 199.

   For information about creating attribute editors, see Chapter 18, "Designing Attribute Editors."

   For information about which attribute editors are appropriate for the data type of this attribute, see "The Attribute Editor Asset" on page 355.

8. (Optional) If you need to override the default ISO character set (ISO 8859-1), click in the **ISO Character Set** field and enter the one you want to use for this attribute.

9. Click **Save**.

## Creating Flex Attributes of Type Blob (Upload Field)

### To create an attribute of type `blob`

1. Complete steps 1 through 4 in the procedure "Creating Flex Attributes: Basic Procedure" on page 318.

2. Click in the **Value Type** field and select **blob**.

3. (Optional) Click in the **Folder** field and enter a path to the directory that you want to store the attribute values in. Note that the value that you enter in this field is appended to the value set as the default storage directory (defdir) for the `MungoBlobs` table.

4. Click in the **Number of Values** field and select **single** or **multiple**, as appropriate. For more information about this field, see "Creating Flex Attributes: Basic Procedure" on page 318.
5. (Optional) If you do not want to use the default input type (a **Browse** button), click in the **Attribute Editor** field and select one of the following:
   - An attribute editor that specifies the `TEXTAREA` input style
   - If your system is configured to use eWebEditPro, an attribute editor that specifies the `EWEBEDITPRO` input style

   For information about attribute editors, see "The Attribute Editor Asset" on page 355.
6. Complete steps 8 and 9 of the procedure "Creating Flex Attributes: Basic Procedure" on page 318.

## Creating Flex Attributes of Type Asset

### To create an attribute of type `asset`

1. Complete steps 1 through 4 in the procedure "Creating Flex Attributes: Basic Procedure" on page 318.
2. Click in the **Value Type** field and select **asset**.
3. Click in the **Asset Type** field and select an option from the drop-down list.
4. Click in the **Mirror Dependency Type** field and select a dependency type.
5. Click in the **Number of Values** field and select either **single** or **multiple** from the drop-down list, as appropriate for the data type that you selected in the **Value Type** field.

   If this attribute is to be used by a flex parent and your data structure allows flex assets to have more than one flex parent, you must select **multiple** because the flex assets who inherit values for this attribute might inherit a value from more than one parent.
6. (Optional) If the number of assets of the type you selected in the **Number of Values** field is more than 20, click in the **Attribute Editor field** and select one. See "Using Attribute Editors" on page 307 for information about appropriate attribute editors.
7. Complete steps 8 and 9 of the procedure "Creating Flex Attributes: Basic Procedure" on page 318.

## Creating Foreign Flex Attributes

If you keep data in another system (a price list, for example) that you also want to use for your flex assets, you can create a foreign attribute that points to the column in the foreign table whose data you want to use as a flex attribute.

### To create a foreign attribute

1. Register the foreign table that contains the data you wish to use as a flex attribute. For information, see "Registering a Foreign Table" on page 225.
2. Complete steps 1 through 6 in the procedure "Creating Flex Attributes: Basic Procedure" on page 318. Note that you **cannot** select either **asset** or **blob** (or url) in the **Value Type** field.

3. (Optional) If you plan to use the CS-Direct Advantage flex asset forms to enter values for the attribute into the foreign table and you do not want to use the default input type for the data type that you selected in the **Value Typ**e field, click in the **Attribute Editor** field and select an appropriate option.
4. Click in the **Editing Style** field and do one of the following:
   - If you want to use the CS-Direct Advantage forms to enter values into this attribute's fields for the flex assets that use it, select **local**.
   - If you do not want users to be able to write values to this table through the CS-Direct Advantage forms, select **external**.
5. Click in the **Storage Style** field and select **external** from the drop-down list.
6. Click in the **External ID** field and specify the name of the column that serves as the primary key for the table that holds this foreign attribute, that is, the column that uniquely identifies the attribute.
7. Click in the **External Table** field and enter the name of the table that stores this attribute.
8. Click in the **External Column** and enter the name of the column in the table specified in the **External Table** that holds the values for this attribute.
9. Complete steps 8 and 9 of the procedure "Creating Flex Attributes: Basic Procedure" on page 318.

## Step 5: (Optional) Create Flex Filter Assets

This section contains general instructions for creating a flex filter asset whose functionality is defined by one of the default Content Server filter classes. For information about flex filter classes, and detailed instructions on creating flex filter assets, see Chapter 17, "Flex Filters."

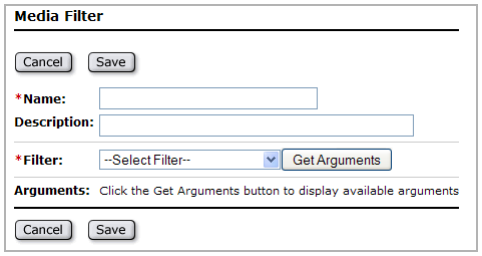
Before you can create flex filter assets, the flex attributes that you plan to use as the input and output attributes must already be created. If the appropriate flex attributes do not exist yet, create them. Note the following requirements:

- For flex filters that use the Document Transformation filter type, the input and output attributes must be of type blob.
- For any flex filter, the input attribute, output attribute, and flex filter asset must all belong to the same flex family.

### To create a flex filter asset

1. If the Content Server interface is not open, log in and select the appropriate site.
2. From the start menu options, click **New**.
3. In the list of asset types, select the type of filter you wish to create.

The “New” filter form is displayed:




4. In the “New” form, fill in the following fields:
   - **Name** – Enter a unique name for this filter asset.
   - **Description** – Enter a brief description summarizing the filter’s function.
   - **Filters** – Select the filter class that will define the functionality of the flex filter asset you are creating. By default, the filter options are “Doc-Type,” “Document Transformation,” “FieldCopier,” and “ThumbnailCreator.”

   **Note**

   If custom filter classes have been created for your system, they will also appear in this list. If you wish to create a custom flex filter class, see “Defining a Flex Filter Class and Creating a Flex Filter Asset” on page 337.

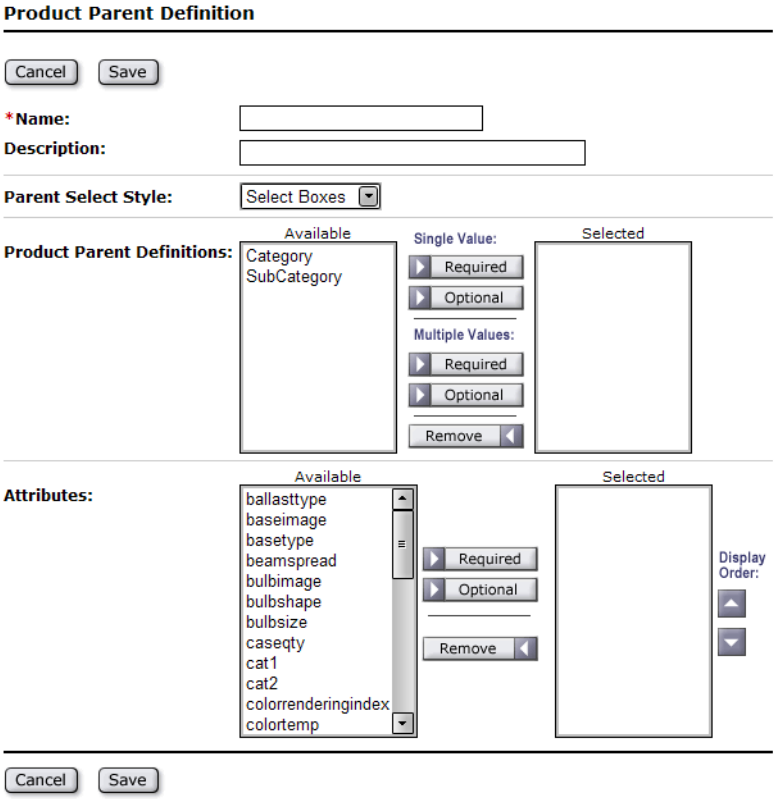
5. Click **Get Arguments**. In the “Arguments” field, specify the input and output arguments for the flex filter asset. Click **Add** to add the arguments to the filter.
6. Click **Save**.

## Step 6: Create Parent Definition Assets

Complete the following steps:

1. If the Content Server interface is not open, log in and select the appropriate site.
2. Click **New** and select the name of your product definition asset from the list of shortcuts.

   The **New** form appears. For example, this is the **New** form for the GE Lighting sample site product parent definition asset type:




3. Click in the **Name** field and enter a name of up to 64 characters.
4. Click in the **Description** field and enter a short, descriptive phrase that describes the parent definition.
5. Click in the **Parent Select Style** field and determine how flex parents that use this definition will be selected on the parent asset forms. Do one of the following:
   - If the number of parents of this type will be small, choose **Select Boxes**. Then, all the parents of this type will be displayed as options in a drop-down field on the flex asset forms.
   - If the number of parents of this type will be large, choose **Pick From Tree**. Then, when you select a parent of this type on the flex asset form, you select it from the tree on the tab that displays your catalog data. For example, on the GE Sample site, the catalog data is displayed in a tree on the **Catalog** tab.

6. Select a parent definition from the **Available** list. For information about selecting parent definitions, see "Determining Hierarchical Place" on page 309.
7. Click the appropriate arrow button, as described in this table:

| Button in parent definition form | Creates a field in the New parent form that does the following: |
|---|---|
| Single Value (Required) | Forces you to select one parent for the field. |
| Single Value (Optional) | Allows you to select only one parent for the field. |
| Multiple Value, (Required) | Forces you to select at least one parent asset for the field. |
| Multiple Value (Optional) | Allows you to select more than one parent asset for the field. |

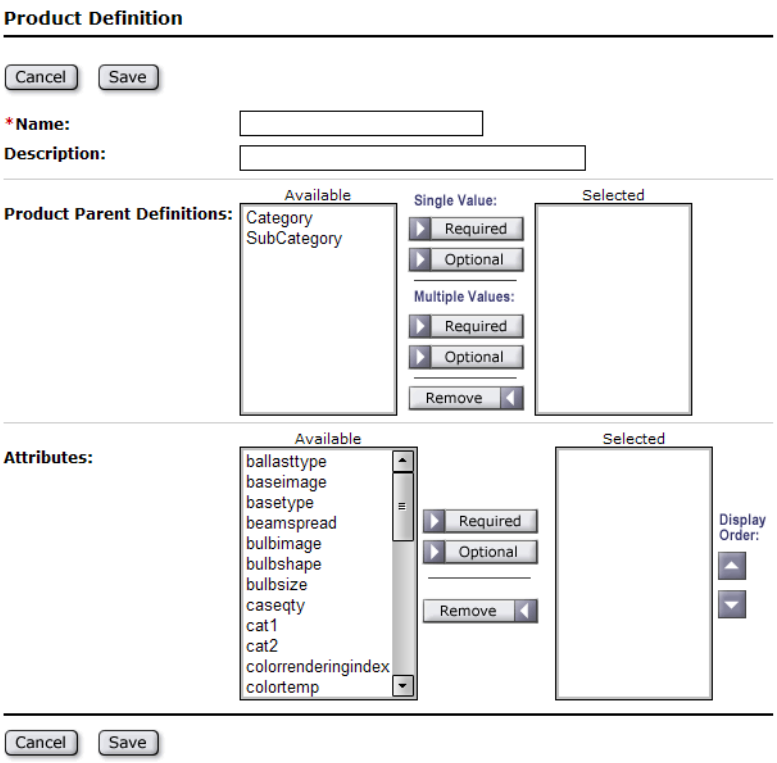
   CS-Direct Advantage moves the parent definition from the **Available** list to the **Selected** list.
8. Repeat steps 6 and 7 as many times as necessary. Remember that the corresponding **New** parent form will include a field for each item that you select in the **Available** list on this parent definition form.
9. In the **Attributes** section, select the appropriate attributes. Note that if you are going to assign a flex filter asset to this parent definition, you must include the input and output attributes that the flex filter uses.
10. Do one of the following:
    - Click the **Required** button to specify that the attribute is required, that is, all flex parents created with this definition must have a value for this attribute.
    - Click the **Optional** button to specify that the attribute is optional.
11. (Optional) If you did not select the attributes in the order in which you want them to appear on the parent form for parents of this type, use the arrow buttons to the right of the **Selected** box to order them.
12. (Optional) In the **Filters** section, select any flex filter assets that are appropriate for this parent definition.
13. Click **Save**.
14. Repeat this procedure for each parent definition asset that you need to create.

## Step 7: Create Flex Definition Assets

Complete the following steps:

1. If the Content Server interface is not open, log in and select the appropriate site.
2. Click **New** and then select the name of your flex definition asset type from the list of shortcuts.

The **New** form appears. For example, this is the **New** form for the GE Lighting sample site product definition asset:




3. Click in the **Name** field and enter a name of up to 64 characters.
4. Click in the **Description** field and enter a short, descriptive phrase that describes the parent definition.
5. Select a parent definition from the **Available** list. For information about selecting parent definitions, see "Determining Hierarchical Place" on page 309.
6. Click the appropriate arrow button, as described in the following table:

| Button in flex definition form | Creates a field in the New flex asset form that does the following: |
|---|---|
| Single Value (Required) | Forces you to select only one parent in the field. |
| Single Value (Optional) | Allows you to select only one parent in the field. |
| Multiple Value (Required) | Forces you to select at least one parent asset in the field. |
| Multiple Value (Optional) | Allows you to select more than one parent asset in the field. |

CS-Direct Advantage moves the parent definition from the **Available** list to the **Selected** list.

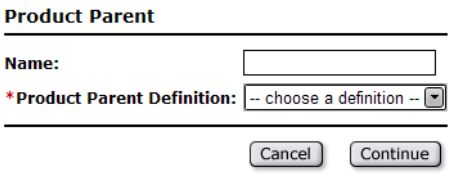
7. Repeat steps 5 and 6 as many times as is necessary. Remember that the corresponding **New** flex asset form will include a field for each item that you select in the **Available** list on this flex definition form.
8. In the **Attributes** section, select an attribute. Note that if you are going to assign a flex filter asset to this flex definition, you must include the input and output attributes that the flex filter uses.
9. Do one of the following:
   - Click the **Required** button to specify that the attribute is required; that is, that all flex assets created with this definition must have a value for this attribute.
   - Click the **Optional** button to specify that the attribute is optional.
10. (Optional) If you did not select the attributes in the order in which you want them to appear on the **New** and **Edit** forms for flex assets created with this definition, use the arrow buttons to the right of the **Selected** box to order them.
11. (Optional) In the **Filters** section, select any flex filter assets that are appropriate for this flex definition.
12. Click **Save**.
13. Repeat this procedure for each flex definition that you need to create.

## Step 8: Create Flex Parent Assets

### To create flex parent assets

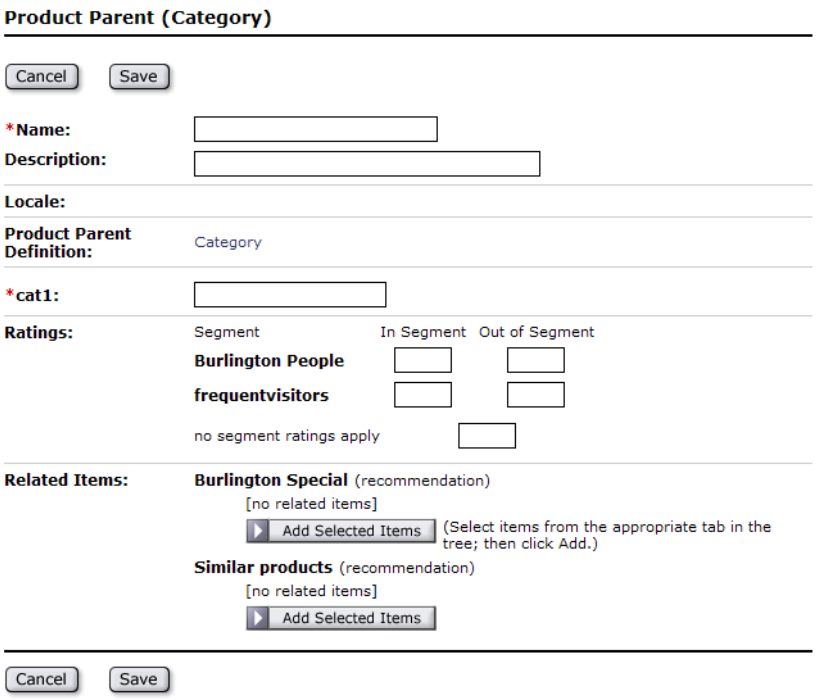
1. If the Content Server interface is not open, log in and select the appropriate site.
2. Click **New** and then select the name of your flex parent asset type from the list of shortcuts.

   The **New** form appears. For example, this is the **New** form for the GE Lighting sample site product parent asset:




3. Click in the **Parent Definition** field and select one from the drop-down list. The definition you select formats the next form, the form you fill out to define this parent asset.

The second **New Parent** form appears.




4. Click in the **Name** field and enter a name of up to 64 characters.
5. The fields displayed in the **Parent** section of the form depend on the parent definition you chose for this parent. The parents that you select in these fields are the grandparents of any flex assets that have the parent you are creating in this procedure. If you do not select any parents (grandparents), the parent you are creating is a top-level parent in the tree tab that displays your flex assets.

   Note the following about the kinds of fields that might appear in this section:

   - If there is an (S) next to a field, you can select one grandparent from the drop-down list.
   - If there is an asterisk (*) and an (S) next to the field, you must select one grandparent from its drop-down list.
   - If there is an (M) next to a field, you can select more than one grandparent from its drop-down list.
   - If there is both an asterisk (*) and an (M) next to a field, you must select at least one grandparent from its drop-down list.
6. In the attributes section of the form, fill in the appropriate values for this parent. If a field has an asterisk (*) next to it, it is a required field.

   The fields displayed in this section are based on the parent definition you chose for this parent. The values that you enter into these fields are inherited by any flex assets that have this parent asset as their parents.
7. Click **Save**.

CS-Direct Advantage writes the new parent to the database. All the information other than the attribute values are written to the *FlexParent*, *FlexParent*_AMap, and *FlexParent*_Extension tables, where *FlexParent* represents the internal name of your flex parents. The attribute values are written to the *FlexParent*_Mungo table.

## Step 9: Create and Assign Asset Type Icons (Dash Interface only)

When a user performs a search or runs a tag in the Dash interface, the list of returned assets displays, for each asset, an icon representing the asset's type in the "Type" column. (Hovering the mouse cursor over the icon will display the asset type.)

You must create and assign icons that uniquely identify the members of your new flex family by completing the steps below.

1. For each flex family member, do the following:
   a. Create a GIF image no larger than 20x20 pixels representing the asset type.
   b. Name the file using the syntax *assetType*.gif. The file name determines the asset type for which the icon will be displayed. The name is case-sensitive.
   c. Place the file in the following directory:

      <cs_app_dir>/Xcelerate/OMTree/TreeImages/AssetTypes/

      where <cs_app_dir> is the directory of the deployed CS application on your application server.
2. Restart your application server for the icons to appear in the Dash interface.

## Step 10: Code Templates for the Flex Assets

Creating your flex asset definitions and coding the templates for the flex assets that use those definitions is an iterative process. Although you need to create definitions and flex assets before you can create templates for your flex assets, it is likely that you will discover areas that need refinement in your data design only after you have coded a template and tested the code.

For information about coding elements for your templates, see Chapter 23, "Creating Template, CSElement, and SiteEntry Assets" and Chapter 27, "Coding Elements for Templates and CSElements."

## Step 11: Test Your Design (Create Test Flex Assets)

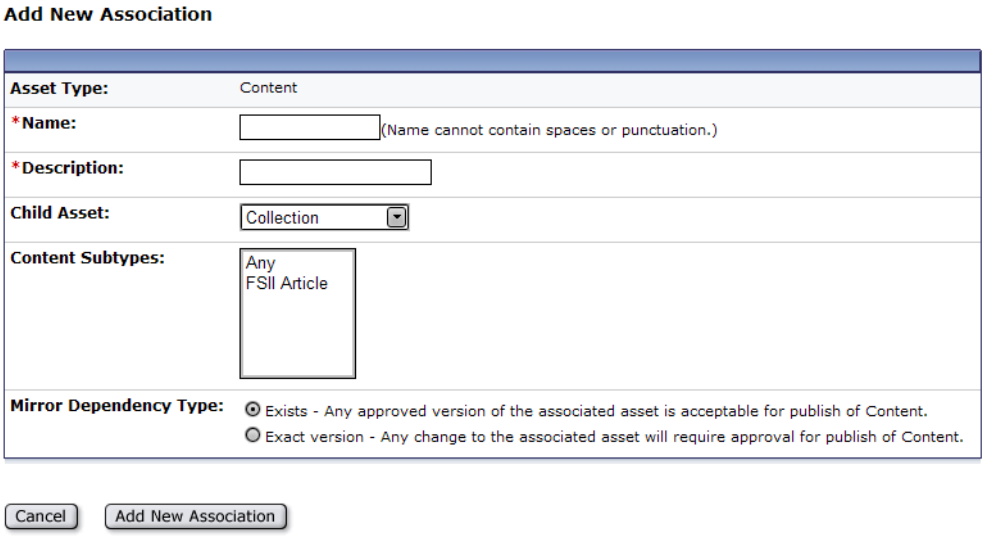
To thoroughly test your design, you must create some flex assets so that you can examine where they appear on the tree, what their forms look like, how long it takes to load their forms, and so on.

For information about creating new assets, see the *Content Server Advanced Interface User's Guide*.

## Step 12 (Optional): Create Flex Asset Associations

In most cases, you should use a flex asset's attributes to form associations. In the rare case that your associations must work across flex definitions, create associations between flex assets by completing the following steps:

1. Log in to the Content Server user interface as a user with general administrator privileges.
2. On the **Admin** tab of the tree, click the **Asset Type** node.
3. Click on the plus sign next to asset type you wish to create an association for.
4. Click on the plus sign for the **Asset Associations** node.
5. Click **Add New**. The following screen appears:




6. Click in the **Name** field and enter a name.
7. Click in the **Description** field and enter a description of the association.
8. Select a child asset to associate with this asset using the **Child Asset** drop-down select box.
9. Select one or more sub-types using the **Subtypes** field.
10. Choose a dependency type for the associated flex asset using the **Mirror Dependency Type** radio buttons.
11. Click **Add New Association** to associate the flex asset types.

## Step 13: Move the Asset Types to Other Systems

When you have finished creating your flex family— which includes creating the new flex asset types with Flex Family Maker, creating the data structure assets (including attribute editors), and coding templates for the flex asset type—you move them to the management and delivery systems.

Then, the system administrators configure the asset types for the management system. They enable revision tracking where appropriate, create workflow processes, create Start Menu shortcuts, and so on.

For information about moving your asset types to the management and delivery systems, see the *Content Server Administrator's Guide*.

# Editing Flex Attributes, Parents, and Definitions

Editing most of the flex asset types requires careful planning because certain edits cause schema changes and **schema changes cause data loss**.

This section presents tips and advice about editing flex family asset types.

## Editing Attributes

Note the following when editing a flex attribute:

- You can change the **Name** without causing a schema change. However, if you are using XMLPost to import flex assets into your Content Server database, you must edit your XMLPost files if you change the name of an attribute.
- You can change the **Description** without causing data loss.
- If you change the data type in the **Value Type** field, you **lose all data** associated with the attribute in the _Mungo table(s) that use this attribute type.
- If the attribute's data type is **asset** and you change the asset type, **all existing data for the attribute is invalid.**
- If you change the **Folder** field for a blob attribute, CS-Direct Advantage will no longer be able to find any existing data for that attribute. If you absolutely must change this value, you need to move the file system to match the new value that you set.
- You can change the **Number of Values** from single to multiple without causing data loss or complications.
- If you change the **Number of Values** from multiple to single, CS-Direct Advantage cannot determine which of the values in any existing rows are the values to keep.
- You can change the **Search Engine** and **ISO Character Set** without causing data loss.

## Editing Parent Definitions and Flex Definitions

Note the following when editing a parent definition or a flex definition:

- You can change the **Name** without causing a schema change. However, if you are using XMLPost to import flex assets into your Content Server database, you must edit your XMLPost files if you change the name of a parent definition.
- You can change the **Description** and the **Parent Select Style** fields without causing data loss.
- If you change the parent selections:
  - Adding parents is allowed
  - Removing parents can cause assets to no longer have valid data.
  - Changing parents from optional to required can cause problems because parents or flex assets who do not have one of the newly required parents are no longer valid.
  - Changing parents from required to optional is allowed.
  - Changing parents from single value to multiple value is allowed.

  - Changing parents from multiple value to single value causes unpredictable results because CS-Direct Advantage can't know which of the previously acceptable multiple values is the one to keep and which ones to remove.
- If you change the attribute selections:
  - Adding optional attributes is allowed.
  - Adding required attributes causes existing parents or flex assets without them to be invalid.
  - Removing attributes causes existing parents or flex assets with such an attribute value to be invalid.

## Editing Parents and Flex Assets

Note the following when editing a flex or parent asset:

- You can change the **Name** without causing a schema change. However, if you are using XMLPost to import flex assets into your Content Server database, you must edit your XMLPost files if you change the name of a parent definition.
- You can change the **Description** without causing data loss.
- If you change parents, CS-Direct Advantage corrects all the inherited attribute values.
- You cannot change the definition that you used to create the parent or flex asset.
- Changing the value of an attribute is allowed. If you change the value of an attribute for a parent, CS-Direct Advantage corrects that attribute for all the assets that inherited it from this parent. Changing the attribute value for a flex asset is allowed.

# Using Product Sets

When you are using CS-Direct Advantage to manage an online catalog, there is a special feature that you can use with product assets called a **product set**. Product sets allow you to group products that are actually the same product except that they are packaged and sold differently.

## What Is a Product Set?

For example, a book is the same book whether it is the paperback version or the hard-cover version. And a soft drink is the same soft drink whether it is sold in individual cans, as a six-pack, in a 2-liter bottle, or a case.

Product sets allow you to group products like these together so that they can be displayed together (in the same form) on the management system, yet remain individual saleable units, identified as such by their SKUs.

The model for the product set feature is as follows:

- The product set is a product parent that takes on the characteristics of a product asset. The product set (parent) has all of the attributes that define the core product.
- The product assets are SKUs. That is, they have only those attributes that describe the packaging or are the unique identifiers for members of the set: the SKU, the bottle size, and so on.
- The product set (parent) has an attribute that marks it as a product set and the value of this attribute is unique among all the product sets. This attribute is called GAProductSet and is a reserved name. The products in the set inherit this attribute and, by this inheritance, are marked as members of that product set (that is, children of that product parent).

## Creating Product Sets

### To create a product set

1. Create a product attribute named `GAProductSet`. This is a reserved name and your attribute name must match it exactly.
2. Create a new product parent definition and select the `GAProductSet` attribute.
3. Create a new product definition and designate that the parents created with the definition that you created in step 2 can be parents of products created with this product definition.
4. Create a new product parent from the definition you created in step 2.
5. Using the product definition that you created in step 3, create the products in the set and designate that the parent that you created in step 4 is their product parent.

Now, when you inspect or edit the product set (product parent), each product (SKU) in the set is listed on the **Product Parent** form, presenting a representation of the product set relationship.

There can be only one **GAProductSet** attribute in the Content Server database. If you have more than one Content Server site and you want to create product sets in more than one site, you must share the **GAProductSet** attribute to the sites that you want to use it in.

Chapter 17

# Flex Filters

This chapter provides information about flex filters. It also contains instructions on defining a flex filter class and creating a flex filter asset.

This chapter contains the following:


- Overview
- Defining a Flex Filter Class and Creating a Flex Filter Asset
- Document Transformation

# Overview

A flex filter performs post-processing operations on flex assets to which that filter is assigned. The filter is called when its associated asset is saved. The filter is not called if the asset's edit operation is canceled.

Multiple flex filters can be assigned to a flex asset. When the asset is saved, its filters perform their designated tasks in the order in which they are defined in the "Filters" section of the asset's definition. The task a flex filter performs is determined by the flex filter class that defines the filter. For example, flex filters that are defined by the FatWire Doc-Type filter class are enabled by that class to determine an asset's MIME-type and to extract a file type from a URL path.

Each flex filter is associated with the attributes that it updates with the data it processes for an asset. These attributes are called *derived attributes*. Derived attributes must be created in the same flex family to which the filter is assigned. However, derived attributes must not be specified in an asset definition because they are strictly a flex filter's output; they are added to an asset only when the asset is saved. When multiple filters are assigned to an asset, none of the filters must be dependent on attributes that are output by any other filters associated with that asset.

When a filter is defined (in the `Filters` table in Content Server's database) it can be applied to any asset type's child definition. (Filters cannot be applied to an asset type's parent definition.) If a filter is created for a specific purpose that is related to a specific asset type, the filter should be associated with only that asset type's child definition. For example, if a filter is designed to process an image type for "Media" assets, it cannot function when applied to text-based assets.

## Flex Filter Classes

**Flex filter classes** implement the functionality of flex filter assets. These classes are listed in the `Filters` table in the Content Server database. When you create a flex filter asset, you select a flex filter class for it.

Content Server delivers the following flex filter classes:

- **Doc-Type –** Extracts components from an asset containing one or more MIME file types and maps each file type to an individually named attribute.
- **Thumbnail Creator –** Converts an image to a thumbnail.
- **Field Copier –** Copies the contents of a system-defined attribute into a user-defined attribute.
- **Document Transformation** – Converts a document from one file type into another by invoking a registered transformation engine (an engine that is specified in the `SystemTransforms` table). The transformation engine functions as a wrapper that forwards calls to a document transformer, which then performs document conversion.

You can create a custom flex filter class by defining it in the Content Server `Filters` database table. For instructions on creating a custom flex filter class, see "Defining a Custom Flex Filter Class," on page 340.

### Doc-Type Filter Class

A Doc-Type filter takes a document and extracts the MIME-type data associated with the file into its individual components. For example, an uploaded file containing text and a GIF photo would be filtered into two generated attributes, one containing TXT formatted data and one containing GIF formatted data.

The Doc-Type class is defined by the following arguments:

- **Attribute to Hold Derived File Name** – (Optional) Enter the flex attribute that stores the output file name.
- **Attribute to Hold Derived File Type** – The file extension is stored in this attribute.
- **Attribute to Hold Derived MIME Type** – (Optional) MIME files can have several types such as plain text, attachment, media file, and so on.
- **Input Attribute Name** – The attribute name stored by Content Server corresponds to the uploaded file name.

### Thumbnail Creator Filter Class

With the thumbnail creator filter, a content provider can upload an original graphic and Content Server will create a new thumbnail sized GIF graphic file.

The Thumbnail Creator class is defined by the following arguments:

- **Input Attribute Name** – The name of the uploaded file.
- Display values for the large version of the thumbnail graphic – "Output Attribute for Main Height," "Output Attribute for Main Width," and "Enter Maximum Pixel Size."
- Attributes that define the thumbnail to be created – "Output Attribute Name," "Output Attribute for Thumb Height," "Output Attribute for Image Aspect," and "Output Attribute for Thumb Width."

## Field Copier Filter Class

The field copier filter copies the contents of a system-defined attribute into a user defined flex attribute.

The Field Copier class is defined by the following arguments:

- **Name** – The name of the system-defined attribute you wish to copy.
- **Value** – The name of the flex attribute into which you are copying the system-defined attribute's value.

Figure 7, on page 400 illustrates an advanced example of how to implement a field copier filter, using the "Media" flex family of the FirstSite II sample site. The purpose of the field copier filter in this example is to categorize image assets by the names of their parent assets.

## Document Transformation Filter Class

The Document Transformation filter class is defined by the following arguments:

- **Document Transformer Name –** The name of a registered transformation engine exactly as it is listed in the `SystemTransforms` table. By default, four values are listed in the `SystemTransforms` table:
  - **CS: Convert to HTML** – Initiates the conversion of binary files to HTML.
  - **CS: Convert to HTML fragment –** Initiates the conversion of binary files into files with HTML coding but without the header tags.
  - **CS: Convert to XML** – Initiates the conversion of binary files to XML.
  - **CS: Convert to Raw Text –** Initiates the conversion of a binary file to TXT format.

**Note**

The following document transformer is available with Content Server:

`com.fatwire.transformer.tika.DocumentTransformerImpl`

The document transformer is coded to convert documents to raw text files when it is invoked by the **CS: Convert to Raw Text** engine. Converting to any other file type requires writing a document transformer for that file type, registering the corresponding transformation engine (unless it is already registered), and registering the document transformer with Content Server. For information about implementing default and document transformation solutions, see "Document Transformation," on page 346.

- **Input Attribute Name –** The name of the flex attribute whose contents are to be converted by the flex filter. For the Document Transformation filter, the input attribute must be of type `blob` because it expects to find a file in that attribute.
- **Fail on Transform Error** – Choose whether you would like the system to display an error message if the transformation does not complete properly.

- **Output Attribute Name –** The name of the flex attribute that stores the results of the document transformation. For the Document Transformation filter, the output attribute must be of type `blob` because it stores the results of the transformation as a file.

  The data stored in the output attribute (field) is read-only because it is derived from the data in the input attribute. This data is regenerated from the source data in the input attribute each time the asset is saved.
- **Output Document Extension –** The file extension to be assigned to the resulting file. Enter a document extension appropriate to the selected **Document Transformer Name**. For example, when you specify that the document transformation engine is **CS: Convert to HTML**, the document extension must be either **htm** or **html**.

## Flex Filter Assets

**Flex filter assets** are defined by all of the following criteria:

- A flex filter class registered in the `Filters` table
- The information that is passed to that filter class (through arguments). These arguments specify which data to use, any constraints on the filter's action, and where to store the results of the filtering process when the asset is saved.

When you create a flex filter asset, you select the flex filter class to be used, then enter values for the arguments that the filter class needs in order to perform its action.

After you create a flex filter asset, you assign it to the appropriate flex definition assets. Then, whenever content providers save a flex asset of that definition, the filter automatically performs its assigned action. For detailed instructions, see "Creating Flex Filter Assets," on page 341.

# Defining a Flex Filter Class and Creating a Flex Filter Asset

To use a flex filter you must first define a custom flex filter class in the Content Server database. You then activate the custom flex filter class by creating a flex filter asset whose functionality is defined by that class.

These are your basic steps:

1. Create the Java class that provides the implementation code for the custom flex filter class. For instructions, see "Implementing a Flex Filter Class," on page 338.
2. Define the new flex filter class in the `Filters` table in the Content Server database. For instructions, see "Defining a Custom Flex Filter Class," on page 340.
3. Create a flex filter asset that is defined by the custom flex filter class:
   a. Create the attributes that will be referenced in the filter's arguments.
   b. Create the flex filter asset and assign it to the desired flex filter class.
   c. Add the new filter asset to a child definition.
   d. Re-save all related assets associated with the definition to which you added the filter asset.

   For instructions, see "Creating Flex Filter Assets," on page 341.

## Implementing a Flex Filter Class

To implement a custom flex filter create a new Java class for it by extending the **AbstractFlexFilter** class. This filter class contains all default functionality required to handle filter requests. A working example of a custom flex filter class that extends the AbstractFlexFilter class, and whose purpose is to access a flex attribute and set a derived attribute value, is provided by the source file `SampleFlexFilter.java`.

A flex filter's functionality is implemented by parameters called abstraction interfaces. Since flex filters only need to know about certain aspects of the assets they are filtering, abstraction interfaces provide filters with access to only the asset information necessary for them to perform their function on a given flex asset. Table 9 lists the abstraction interfaces required to implement a flex filter, and the asset information each abstraction interface passes to the filter. For detailed information about the various abstraction interfaces used to implement a custom flex filter class, see the *Content Server Javadoc*.

**Table 9:** Abstraction Interfaces (`com.openmarket.gator.interfaces` package)

| Abstraction Interface | Description |
|---|---|
| `IFilterEnvironment` | Provides methods to obtain information about the environment that is supporting the filter. |
| `IFilterableAssetInstance` | Provides methods to manipulate the asset that is being filtered. |
| `IFilterDescription` | Provides methods to describe all the potential derived attributes that will be modified by the filter during its execution. |
| `IFilterDependencies` | Provides methods to log the dependencies against the asset instance to be filtered. A dependency refers to another asset by type and identifier that shares a relationship with the asset to be filtered. When a dependency is declared, it is either exact or exists. |

**Note**

Standard asset attributes can be obtained by the `IFilterableAssetInstance.get` method. For example, to get the standard asset description, you would add the line:

```
String description = instance.get(description);
```

When building the Java code for the new flex filter class, define a constructor with a single `FTValList` parameter (located in the `COM.FutureTense.Interfaces` package). This parameter provides a list of arguments obtained from the filter's definition in the `Filters` database table. These arguments are passed to the filter in the form of key/value pairs. If there are no predefined arguments for the filter class in the `Filters` database table, the `FTValList` is null.

## Extending the 'AbstractFlexFilter' Class

The **AbstractFlexFilter** class can be extended to build your implementation of a flex filter class. This class is located in the `com.openmarket.gator.flexfilters` package. When you create the new flex filter's Java class, you can call the required methods necessary for your filter's functionality from the AbstractFlexFilter class' java code. This simplifies the amount of code necessary to create a custom implementation.

When a method is called, it is provided with the argument `String filterIdentifier`, which is the asset identifier for the filter. When you associate the same filter with different flex asset families, the filter identifier reflects the filter definition for the associated family. This argument is useful when you are implementing a filter that will be used by different asset families in order for the filter to know which flex family association is being used at the moment the filter is invoked.

For detailed information about the various abstract methods used to implement a flex filter class, see the *Content Server Javadoc*.

### Required Abstract Methods

The following abstract methods are required by all flex filter implementations:

```
public void filterAsset(IFilterEnvironment env,
      String filterIdentifier,
      FTValList filterArguments,
      IFilterableAssetInstance instance)
      throws AssetException;
```

These lines are the main method to process asset post processing when a new or pre-existing asset is saved. This method is not called if the edit is cancelled. It does the work that represents the filter's purpose. A list of arguments (`FTValList`) is provided so the filter can obtain input and/or output attribute definitions, and any other information valid to the filter. The `filterArguments` list is defined when the filter is created in the Content Server interface.

```
public FTValList getLegalArguments(IFilterEnvironment env,
      String filterIdentifier)
      throws AssetException;
```

These lines are the method that is called to return a list of legal filter arguments. The Content Server interface will call this during filter creation or editing to populate the combo box after selecting the filter and pressing the **Get Arguments** button.

### Optional Abstract Methods

The following abstract methods can be used to override those within the AbstractFlexFilter class. These methods are optional because the default implementations provided by the AbstractFlexFilter class are usually sufficient for most filters:

```
public void describeDerivedAttributes(IFilterEnvironment env,
      String filterIdentifier, FTValList filterArguments,
      String defTypeName, String parentDefTypeName,
      IFilterDescription descriptionObject) throws AssetException
```

This method describes all the potential derived attributes, group affinities, and recommendations that the filter might set. When the filter plans to output to attributes this method must identify the attributes that will be modified. This is called whenever the flex asset is viewed, to anticipate the editing of the asset.

```
public void getDependencies(IFilterEnvironment env,String
   filterIdentifier,
   FTValList filterArguments, String assetTypeName, String
      parentTypeName,
   IFilterDependencies filterdeps) throws AssetException
```

This method is called to describe the filter's asset dependencies. Filter dependencies are set to either **exists** or **exact**.

```
public String[] getArgumentLegalValues(IFilterEnvironment env,
   String filterIdentifier, String argumentName) throws
      AssetException
```

This method is called to return a list of acceptable values for a specified argument. If the return list is null then any value is accepted. This method is called by the Content Server Advanced interface when a filter asset is being created or edited to validate the argument values that are specified for the filter asset.

## Defining a Custom Flex Filter Class

This section provides instructions for defining a custom flex filter class in the `Filters` table in the Content Server database. In this example, we create a flex filter class named **CustomFilter**.

### To define a custom flex filter class

1. Copy the `.jar` or `class` file containing the implementation code for the custom flex filter class into the directory that holds the Content Server product `jars`:

   **Note**

   For information about creating a Java class that provides the implementation code for your custom flex filter class, see "Implementing a Flex Filter Class," on page 338.

   - For WebLogic: `app-server-install-dir/bea/path-to-domain/domain-name/applications/WEB-INF/lib`
   - For WebSphere: `WebSphere-Installation-Directory/InstalledApps/WEB-INF/lib`
   - For Sun ONE: `domain-name/application-name/applications/J2ee-apps/ContentServer/cs_war/WEB-INF/lib`
   - For Oracle Application Server: use the Oracle Admin console to deploy the `.jar` file.

2. Open CS-Explorer and add a row to the `Filters` table for the new filter class:
   a. In the tree, expand the **Tables** node, and then select the `Filters` table.
   b. Select **File > New > Record**.
   c. Define the filter in the database by filling in the following columns:
      - **name** – Enter the name of the filter as it will be displayed in the Content Server interfaces.
      - **description** – (Optional) Enter a short summary about the purpose of the filter class.

- **classname** – Enter the exact classname of the filter class' implementation (for example, `com.fatwire.firstsite.filter.SampleFlexFilter`). This name must be available for loading in the Content Server classpath.
- **args** – (Optional) Enter the input and output arguments for the filter. Argument key/value pairs are delimited by an ampersand (&) character. (for example, `arg1=argument1&arg2=argument2`). These arguments are passed to the filter constructor and their use is left up to the filter's developer.

> **Note**
>
> If you do not define the flex filter class' input and output arguments in the `Filters` table, you can define the arguments in the flex filter class' code. For information, see "Implementing a Flex Filter Class," on page 338.

**d.** Select **File > Save**.

Your new filter entry will look similar to the following:

| name | description | classname | args |
|---|---|---|---|
| CustomFilter | Sample Flex Filter | com.fatwire.firstsite.filter.SampleFlexFilter | Input custom string=argument1&Output custom string=argument2 |
| DocType | Extracts Mime Type from Document | com.fatwire.firstsite.filter.DocType | |
| Document Transformation | Use a registered document transformation engine | com.openmarket.gator.filters.DocTransformation | |
| FieldCopier | Copies Base Fields to Attributes | com.fatwire.firstsite.filter.FieldCopier | |
| ThumbnailCreator | Creates Thumbnail | com.fatwire.firstsite.filter.Imaging | |

The filter class is now displayed as an option in the **Filter** drop-down list in the "New" and "Edit" forms of filter assets.

## Creating Flex Filter Assets

This section provides instructions for creating a flex filter asset. In this example, we create a flex filter asset named **FSII_CustomFlexFilter** for the "Media" flex family of the FirstSite II sample site.

Before you can create a filter asset, the flex attributes that you want to use as the input and output attributes must exist. To ensure that the values of preexisting attributes are not overwritten, you can create new attributes to be referenced by the arguments for your custom flex filter. For this example, we created two "Media" attributes of type `string` – **FSII_CustomInput** to be used as the input attribute and **FSII_CustomOutput** to be used as the output attribute.

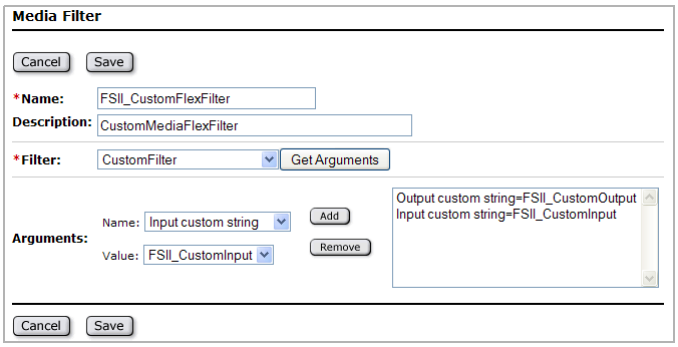
### To create a flex filter asset

1. Log in to the Advanced interface as a general administrator (`fwadmin/xceladmin` in this example), and select the site for which you wish to create a flex filter asset (FirstSite II sample site in this example).
2. Create the attributes (if not already defined) that will be referenced by the filter's input and output arguments. Note the following requirements:
   - For flex filters that use the Document Transformation filter class, the input and output attributes must be of type blob.
   - For any flex filter, the input attribute, output attribute, and flex filter must all belong to the same flex family.

For instructions on creating flex attributes, see "Step 4: Create Flex Attributes," on page 317.

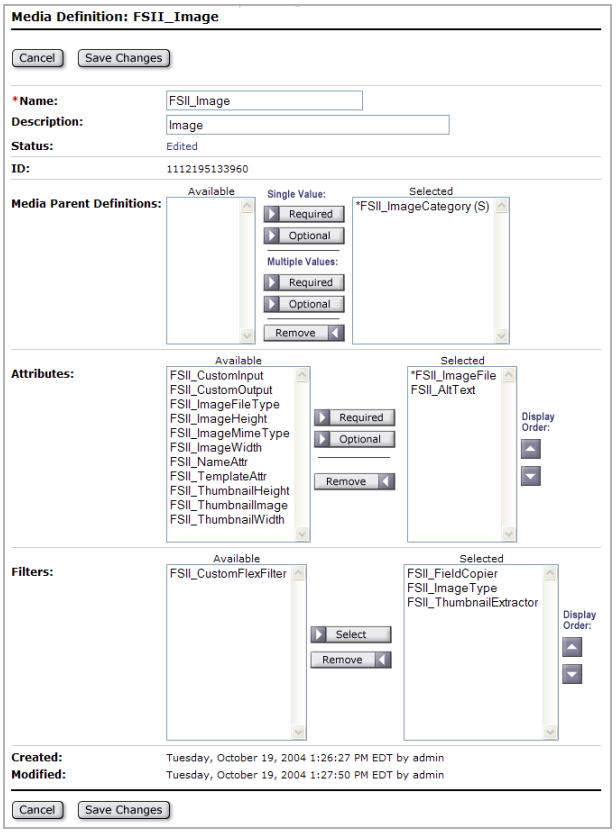
3. Create a flex filter asset for the associated flex family ("Media" flex family in this example):
   a. From the start menu options, click **New**.
   b. In the list of asset types, select the type of filter you wish to create. In this example, **New Media Filter**.

      The "New" filter form is displayed.
   c. Fill in the following fields:
      1) In the **Name** field, enter a unique name for this filter (in this example, we use **FSII_CustomFlexFilter**).
      2) In the **Description** field, enter a brief description summarizing the filter's function.
      3) In the **Filters** drop-down list, select the filter definition that matches the name you assigned to the custom filter (in this example, we select **CustomFilter**). Then click **Get Arguments**.
      4) In the "Arguments" field, specify the input and output arguments for the flex filter asset. Click **Add** to add the argument(s) to the filter.




   d. Click **Save** to save the filter.
4. Find the desired child definition to which you wish to add the new filter asset (in this example, we add the filter to the "Media" child definition **FSII_Image**).
   a. In the start menu options, click **Search**.
   b. In the list of asset types, select the desired type of asset definition to which you wish to add the filter (**Find Media Definition** in this example).
   c. In the search field, enter the name of the definition to which you wish to add the filter (**FSII_Image** in this example).
   d. Click **Search**.
   e. In the list of search results, navigate to the desired definition and click its **Edit** (pencil) icon (**FSII_Image** in this example).

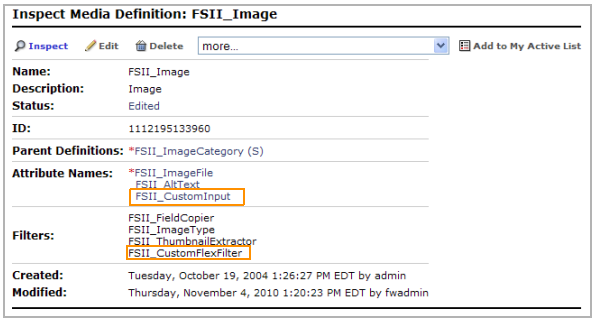
The “Edit” form of the definition opens:



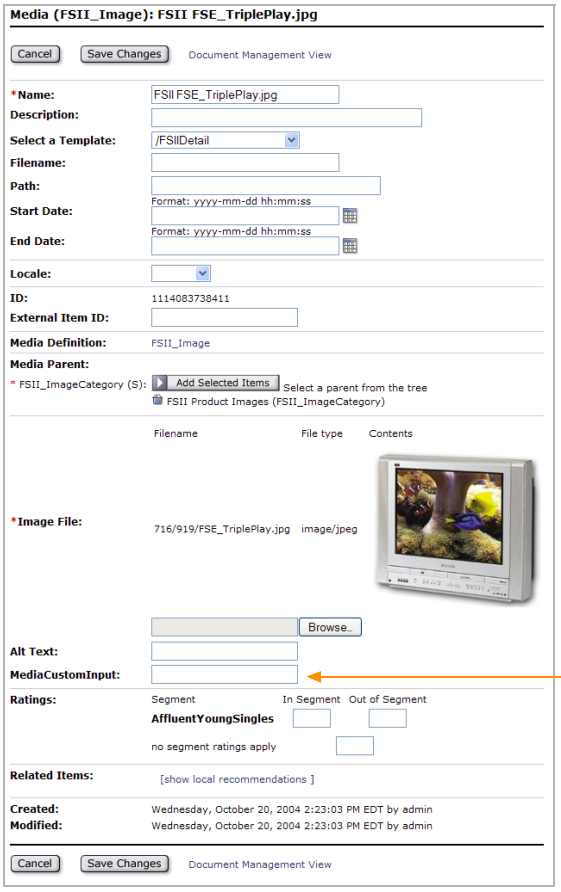

5. In the “Edit form, add the filter and input argument to the definition:
   a. In the “Attributes” field, highlight the input attribute in the **Available** list box and move it to the **Selected** list box (**FSII_CustomInput** in this example).
   b. In the “Filters” field, highlight the desired flex filter in the **Available** list box and move it to the **Selected** list box (**FSII_CustomFlexFilter** in this example).

> **Note**
>
> Add the new filter after any other filters that will create or modify attributes which the filter you are adding depends on or shares in common.

   c. Click **Save Changes**.

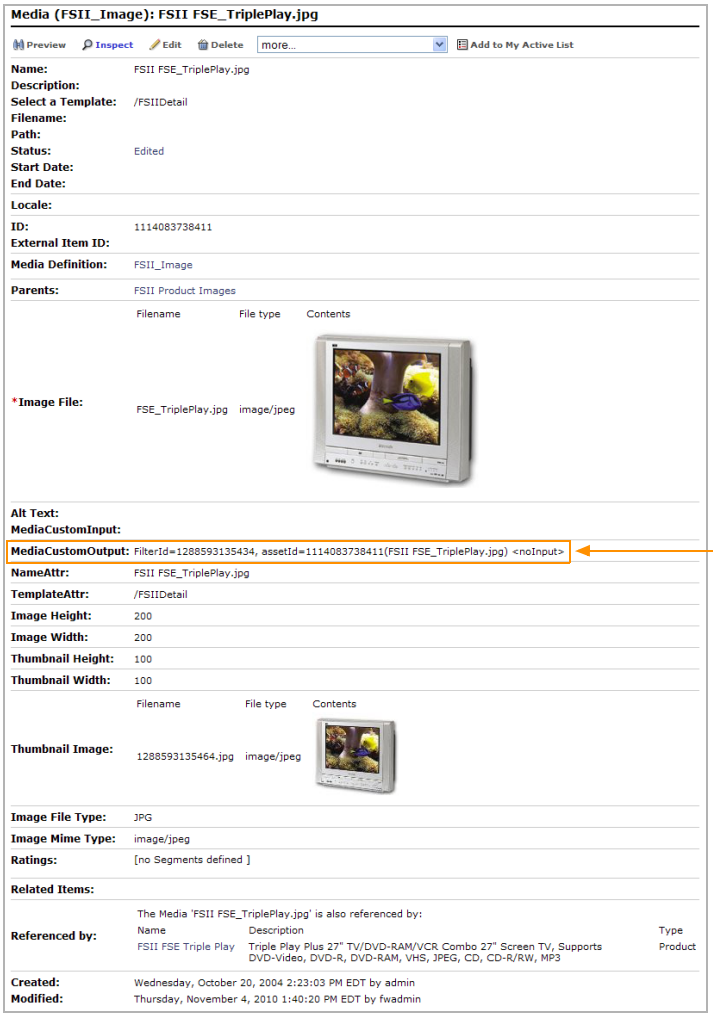


6. Find and re-save all preexisting assets associated with the definition to which you added the filter. This enables the filter to populate the output attribute (**FSII_CustomOutput**) with the derived value from the input attribute (**FSII_CustomInput**). For example:
   a. In the start menu options, click **Search**.
   b. In the list of asset types, select the type of asset associated with the definition you modified in step 5 (**Find Media** in this example).
   c. In the search screen, click **Search**.
   d. In the list of search results, navigate to and select the **Edit** (pencil) icon of any asset (**FSII FSE_TriplePlay.jpg** in this example).




Input attribute that you added to the definition in step 4.

**e.** Click **Save Changes** to re-save the asset.




Output attribute containing the derived `string` value of the input attribute.

The above image is the result of editing a flex asset that invokes several filters, including the filter you created in this section. By default, “Media” assets call the “FSII_FieldCopier,” “FSII_ImageType,” and “FSII_ThumbnailExtractor” filters. In this example, the “Media” asset also calls the **FSII_CustomFlexFilter** filter, which takes the value of the input attribute (**MediaCustomInput**) and inserts a derived string value into the output attribute (**MediaCustomOutput**).

# Document Transformation

Implementing a Document Transformation flex filter requires the following components:

- A transformation engine that is registered in the `SystemTransforms` table and named to indicate the target file type; for example, `CS:Convert to Raw Text`.
- A document transformer, which is a custom class that performs document conversion (for example, converting binary files to raw text files). The document transformer class implements the following interface: `com.fatwire.transformer.common.DocumentTransformer` interface
- The `transformer-formats.xml` file, which is used to associate the transformation engine with the document transformer. The file, located in Content Server's `WEB-INF/classes` folder, specifies the target file type and the document transformer.

When the Document Transformation flex filter is invoked, the transformation engine functions as a wrapper. The engine forwards calls (via the `transformer-formats.xml` file) to the document transformer, which then performs file conversion.

## Default Solution

Content Server provides a default solution that can be used to convert documents to raw text files. The default components are:

- The `CS:Convert to Raw Text` transformation engine, registered in the `SystemTransforms` table.
- A document transformer named `com.fatwire.transformer.tika.DocumentTransformerImpl`, which is coded to output raw text files once it is invoked by the `CS:Convert to Raw Text` engine.
- The `transformer-formats.xml` file, which is configured to associate the `CS:Convert to Raw Text` engine with the document transformer class named above.

Using the default solution requires you to implement a corresponding Document Transformation flex filter, which makes the transformation engine accessible from Content Server's interface as a document transformation option.

## Custom Solutions

If you need a document transformation solution other than the default solution described above, you will have to create and customize the following components:

1. Write and deploy a document transformer for the target file type.
2. Register the transformation engine for the target file type.
3. Configure the `transformer-formats.xml` file to specify the document transformer and the target file type. (The xml file supports multiple document transformers.)
4. Implement the Document Transformation flex filter as described in this chapter.

For detailed steps, see "Customizing Document Transformation," on page 347.

## Customizing Document Transformation

Content Server provides a default document transformer, `com.fatwire.transformer.tika.DocumentTransformerImpl`, which is coded to output raw text files once it is invoked by the `CS:Convert to Raw Text` engine. If the target file type must be other than raw text, you will have to complete the following steps to create and register the flex filter's supporting components:

Before implementing a Document Transformer flex filter

- Writing and Deploying a Document Transformer
- Registering the Transformation Engine
- Registering the Document Transformer

### Writing and Deploying a Document Transformer

1. Write an implementation of the `com.fatwire.transformer.common.DocumentTransformer` interface. You will implement the following methods:

   ```
   public String getOutputDocument(String filename,
      TransformerFormat outputformat);
   ```

   and

   ```
   public String getOutputDocument(String filename,
      String inputFileExt,TransformerFormat outputformat);
   ```

   You can return `null` in the following method, as this method has been deprecated:

   ```
   public InputStream getBytesAsStream(String filename,
      TransformerFormat outputformat);
   ```

2. Copy the document transformer's `jar` or `class` file to the Content Server web application `lib` folder or `classes` folder.
   - For WebLogic: `app-server-install-dir/bea/path-to-domain/domain-name/applications/WEB-INF/lib`
   - For WebSphere: `WebSphere-Installation-Directory/InstalledApps/WEB-INF/lib`
   - For Sun ONE: `domain-name/application-name/applications/J2ee-apps/ContentServer/cs_war/WEB-INF/lib`
   - For Oracle Application Server: Use the Oracle Admin console to deploy the `.jar` file.
3. Continue to the next step, "Registering the Transformation Engine."

## Registering the Transformation Engine

**Note**

If your document transformer is written to output files other than HTML, HTML fragment, or XML, skip to step 1 below this note. Otherwise, continue reading this note.

**Using a Default Transformation Engine**

Content Server provides the following transformation engines, in addition to CS:Convert to Raw Text:

- CS: Convert to HTML
- CS: Convert to HTML fragment
- CS: Convert to XML

All of the engines are registered by default in the `SystemTransforms` table. If your document transformer is written to output files of type HTML, or HTML fragment, or XML, use the corresponding engine. Do the following:

1. Open CS-Explorer.
2. Select the `SystemTransforms` table.
3. Locate the required engine. For example: `CS:Convert to HTML`
   - Note the value of the engine's `target` field. You will enter this value for `<mime-type>` in the `transformer-formats.xml` file, in the next section, "Registering the Document Transformer."
   - In the `args` field, set the arguments that are appropriate for this transformation engine. For example: `exporttype=HTML`
4. Continue to "Registering the Document Transformer," on page 349.

### To register a transformation engine

1. Open CS-Explorer and add a row to the `SystemTransforms` table for the new transformation engine, as follows:
   a. Select the `SystemTransforms` table and click in the table's workspace.
   b. Select **File > New > Record** and fill in the new row as follows:
      1) In the `name` column, enter the name of the transformation engine. For example: **`ConvertToPDF`**
      2) In the `description` column, enter a description of the engine. For example: **`Convert to PDF`**
      3) In the `target` column, enter `text/<filetype>`. For example: **`text/PDF`**

**Note**

You will use the target value for the `<mime-type>` in the next section, "Registering the Document Transformer."

4) In the `classname` column, enter the engine's default class name: **`com.fatwire.transformer.common.FWTransformer`**

5) In the `args` column, set any arguments that are appropriate for this transformation engine. For example: **`exporttype=PDF`**

c. Save your changes.

2. Continue to the next step, "Registering the Document Transformer."

## Registering the Document Transformer

Register your document transformer in the `transformer-formats.xml` file, located in Content Server's `WEB-INF/classes` folder. The default file is shown below for reference.

**Note**

- In line 3, set the `<mime-type>` to the value of the `target` field for the transformation engine (the value for `target` was set in "Registering the Transformation Engine").
- In line 10, specify the class name of your document transformer (created in "Writing and Deploying a Document Transformer," on page 347).
- If you wish to specify multiple document transformers, repeat lines 6–18 for each transformer. (The value for `<mime-type>` (line 3) determines which document transformer will be invoked by the transformation engine.)

### Default transformer-formats.xml

```
1  <transformer-format>
2  <name>Text Format</name> <!-- name of the output format
                              supported by this repository -->
3  <mime-type>text/plain</mime-type>

4  <file-extension>txt</file-extension>
5  <transformer-options> <!-- number of possible transformers
                            available for this transformation -->
6    <transformer>
7      <name>TIKA</name>
8        <properties>
9           <property>
10             <name>ClassName</name> <!-- name of the
                              transformer class that gets loaded
                              by transformer factory -->


11             <value>com.fatwire.transformer.tika.
                              DocumentTransformerImpl</value>
12          </property>
13          <property>
14             <name>InputFile_Exts</name> <!-- allowed input
                              file extensions..* means all file
                              types supported by tika -->
15             <value>*</value>
```

```
16          </property>
17       </properties>
18   </transformer>
19 </transformer-options>
20</transformer-format>
```

## Next Steps

Once you have created and registered the document transformation components as shown in "Customizing Document Transformation," on page 347, you can implement the corresponding Document Transform flex filters. Use the procedures provided in this chapter.

Chapter 18

# Designing Attribute Editors

An attribute editor specifies how data is entered for an attribute when that attribute is displayed on a “New” or “Edit” form for a flex asset or a flex parent asset in the Content Server interface on the management system.

When you assign an attribute editor to an attribute, it replaces the default input mechanism (style) that would otherwise be used for that attribute. The default input style is based on the data type of the attribute.

Because attribute editors format the input mechanism for attributes, you design your attribute editors as you design your flex attributes. Attribute editors are assets, which means you can use the workflow and revision tracking features to manage them as you do for any other type of asset.

This chapter contains the following sections:


- Overview
- Creating Attribute Editors
- Customizing Attribute Editors
- Editing Attribute Editors

# Overview

There are three parts to an attribute editor, with an optional fourth and fifth:

- The `presentationobject.dtd` file, located in the Content Server installation directory. (Required.) This is the DTD file that defines all the possible input styles (presentation objects) for flex attributes and their style tags.
- The attribute editor asset. (Required.) It holds or points to XML code that provides input options for the attribute it is associated with. You use the style tags defined in the DTD to create this XML code.
- An element that formats the attribute, or, displays an edit mechanism, when that attribute appears in a "New" or "Edit" form. (Required.) This element must be located in the `OpenMarket/Gator/AttributeTypes` directory in the `ElementCatalog` table for CS-Direct Advantage to be able to find it and its name must exactly match the name of the style tag that invokes it from the attribute editor. (See below for more information.)
- An element that formats the attribute value when it appears in an "Inspect" form. (Optional.) This element must also be located in the `OpenMarket/Gator/AttributeTypes` directory in the `ElementCatalog` table.

  The name of the element must use the convention `Display`*`StyleTag`*, where *`StyleTag`* represents and must exactly match the name of the style tag that invokes it from the attribute editor.
- An element that formats the attribute data before it is saved in the database (Optional.) This element must also be located in the `OpenMarket/Gator/AttributeTypes` directory in the `ElementCatalog` table.

  The name of the element must use the convention *`StyleTag`*FlexAssetGather, where *`StyleTag`* represents and must exactly match the name of the style tag that invokes it from the attribute editor.

CS-Direct Advantage provides the following items, by default, to support the development of your attribute editors:

- The `presentationobject.dtd` file. It defines several input styles (presentation objects) that you can use in your attribute editors. This means you do not have to define your own unless the nine that are included do not cover your needs.
- Nine text files with sample XML that you can use to create attribute editor assets. You can cut and paste the sample XML into your attribute editor assets. These files are located in the installation directory under `Samples/Attribute_Editors.`
- Ten display elements that work with the sample XML code for attribute editor assets. They are located in the `OpenMarket/Gator/AttributeTypes` directory in the `ElementCatalog` table.

Remember that attribute editors are not required. When you do not use attribute editors, CS-Direct Advantage uses default input styles for the attributes, based on their data types. For a list of the default styles, see "Default Input Styles for Attributes" on page 199. If the default input styles are sufficient for your attributes, you do not need to create attribute editors.

## The `presentationobject.dtd` File

The `presentationobject.dtd` file defines all of the input types (presentation objects) that you can implement through attribute editors. The default `presentationobject.dtd` file defines nine input style tags and the arguments that they can pass from the attribute editor to the display elements (described in "The Attribute Editor Elements" on page 361).

Following is the entire `presentationobject.dtd` file. It is located in the Content Server installation directory:

```
<!-- PRESENTATIONOBJECT: An editor
-- PRESENTATIONOBJECT defines the presentation object
-- for instances of Gator attribute types. A presentation object
-- defines the properties of an editor for one of the following
-- controls:
--
-- For additional information, refer to
-- com.openmarket.gator.interfaces.IPresentationObject.
--
-->

<!ELEMENT PRESENTATIONOBJECT (TEXTFIELD | TEXTAREA | PULLDOWN |
RADIOBUTTONS | CHECKBOXES | PICKFROMTREE | EWEBEDITPRO | REMEMBER
|PICKASSET)>

<!ATTLIST PRESENTATIONOBJECT NAME CDATA #REQUIRED>

<!-- TEXTFIELD: A text field of a specific width
-- You must specify the x dimension; the maximum number of
-- allowable characters defaults to 256.
-->
<!ELEMENT TEXTFIELD ANY>
<!ATTLIST TEXTFIELD XSIZE CDATA #REQUIRED>
<!ATTLIST TEXTFIELD MAXCHARS CDATA "256">
<!ATTLIST TEXTFIELD BLANKED (YES | NO) "NO">

<!-- TEXTAREA: A text area of a specific size
-- You must specify the x and y dimensions; the wrap style
-- defaults to soft.
-->
<!ELEMENT TEXTAREA ANY>
<!ATTLIST TEXTAREA XSIZE CDATA #REQUIRED>
<!ATTLIST TEXTAREA YSIZE CDATA #REQUIRED>
<!ATTLIST TEXTAREA WRAPSTYLE (OFF | SOFT | HARD) "SOFT">

<!-- PULLDOWN: A pulldown menu with an enumeration of items
-- You can specify zero or more list items; the fontsize
-- defaults to relative fontsize 3.
-->
<!ELEMENT PULLDOWN ((ITEM)* | QUERYASSETNAME)>
<!ATTLIST PULLDOWN FONTSIZE CDATA "3">

<!-- RADIOBUTTONS: Radio buttons with an enumeration of items
```

```
-- You can specify zero or more list items; the fontsize
-- defaults to relative fontsize 3.
-->
<!ELEMENT RADIOBUTTONS ((ITEM)* | QUERYASSETNAME)>
<!ATTLIST RADIOBUTTONS FONTSIZE CDATA "3">
<!ATTLIST RADIOBUTTONS LAYOUT (HORIZONTAL | VERTICAL)
"HORIZONTAL">

<!-- CHECKBOXES: Check boxes with an enumeration of items
-- You can specify zero or more list items; the fontsize
-- defaults to relative fontsize 3.
-->
<!ELEMENT CHECKBOXES ((ITEM)* | QUERYASSETNAME)>
<!ATTLIST CHECKBOXES FONTSIZE CDATA "3">
<!ATTLIST CHECKBOXES LAYOUT (HORIZONTAL | VERTICAL) "HORIZONTAL">

<!-- ITEM: A list item
-- You can specify zero or more characters of text.
-->
<!ELEMENT ITEM (#PCDATA)*>

<!-- SQL: Query to populate list of items
-- You can specify zero or more characters of text. Query must
-- return a 'value' column.
-->
<!ELEMENT QUERYASSETNAME (#PCDATA)*>

<!-- EWEBEDITPRO: EWebEditPro ActiveX widget
-- You must specify the x and y pixel dimensions-->
<!ELEMENT EWEBEDITPRO ANY>
<!ATTLIST EWEBEDITPRO XSIZE CDATA #REQUIRED>
<!ATTLIST EWEBEDITPRO YSIZE CDATA #REQUIRED>

<!-- PICKFROMTREE: An "add from tree" button. -->
<!ELEMENT PICKFROMTREE ANY>

<!-- REMEMBER: The Content Centre version 3.6 remember widget. -->
<!ELEMENT REMEMBER ANY>

<!-- PICKASSET: When the tree is active, it's the "add from tree"
-- button. When the tree is disabled, it's The Content Centre
-- version 3.6 remember widget. -->
<!ELEMENT PICKASSET ANY>
```

## Conventions for the presentationobject.dtd File

If you want to create custom attribute editors other than the ones made possible by default, you must first define an XML input style tag, a `PRESENTATIONOBJECT` tag, in the `presentationobject.dtd` file. To define a new `PRESENTATIONOBJECT` tag, you must do the following:

- Add the new tag (presentation object) to the list in the `<!ELEMENT PRESENTATIONOBJECT ...>` statement.
- Add a `<!ELEMENT ...>` section that defines the new tag (presentation object) and the arguments that it takes. Follow the normal syntax rules for a `.dtd` file and follow the conventions used in the `presentationobject.dtd` file.

# The Attribute Editor Asset

The attribute editor asset either holds XML code or points to an `.xml` file.

That XML code does one thing: if the input type is one that provides options (check boxes, radio options, pull-down lists, and so on), it provides the values of those options.

Although CS-Direct Advantage provides nine text files with sample code that you can use to create new attribute editor assets, it does not provide any attribute editor assets because you need to customize the sample code so that any options are appropriate for your data.

When you create your attribute editors, you can either cut and paste the code from HTML version of this book (samples follow this section) or you can use the text files located in the `Samples` subdirectory of the installation directory on your system.

## The Syntax and the Default Tags

The code in an attribute editor asset has the following basic format:

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT SYSTEM "presentationobject.dtd">

<PRESENTATIONOBJECT NAME="SomeName">
...
...
...

</PRESENTATIONOBJECT>
```

The tag that describes the format of the input style (presentation object) is embedded between the pair of `PRESENTATIONOBJECT` tags and it can have additional nested tags in it. Although the `NAME` attribute is required for the `PRESENTATIONOBJECT` tag, it is not used yet; it is reserved for future use.

The name of any `PRESENTATIONOBJECT` tag that you include in the code for an attribute editor asset must be defined in the `presentationobject.dtd` file. This `.dtd` file has the following `PRESENTATIONOBJECT` tags defined by default:

- `TEXTFIELD`
- `TEXTAREA`
- `PULLDOWN`
- `RADIOBUTTONS`
- `CHECKBOXES`
- `EWEBEDITPRO`

- PICKASSET
- REMEMBER
- PICKFROMTREE (deprecated; use PICKASSET instead)

Note that the PRESENTATIONOBJECT tag that you use in the attribute editor code must exactly match the name of the display element that you want to use for the attribute editor. Therefore, if you decide to define a new tag for a custom attribute editor, the element that you create must use the same name as the tag.

For a description of the elements, see "The Attribute Editor Elements" on page 361. For code samples for attribute editors, read on:

## CHECKBOXES Example

The presentationobject.dtd defines a CHECKBOXES tag—an attribute editor that uses the tag invokes the CHECKBOXES element, which creates a set of check boxes for the attribute.

The CHECKBOXES tag takes the following parameters:

- ITEM or QUERYASSETNAME — the source of the names listed next to the check boxes. To specify the names, use the ITEM parameter. To specify a query asset that obtains the names dynamically from a database table, use the QUERYASSET parameter.

  Note the following:

  - You cannot use a SQL statement — you must use a query asset if you want to use a query.
  - The SQL in the query asset must return a "value" column. For example: select name as value from shippingtype
  - If the data type of the attribute using the attribute editor is "asset", the query must also return the assets' IDs. For example: select name as value, id as assetid from Product where...
- LAYOUT— whether the check boxes should be positioned in a vertical list or spread out in a horizontal row. Valid options are HORIZONTAL or VERTICAL. The default is HORIZONTAL.

The following attribute editor code specifies that the CHECKBOXES element should use the results of a query asset named A Prods for the names of a vertical list of check boxes:

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT SYSTEM "presentationobject.dtd">
<PRESENTATIONOBJECT NAME="CheckBox">
   <CHECKBOXES LAYOUT="VERTICAL">
         <QUERYASSETNAME>A Prods</QUERYASSETNAME>
   </CHECKBOXES>
</PRESENTATIONOBJECT>
```

For example code that shows the use of the ITEM parameter, see "PULLDOWN Example" on page 358.

### Notes About Data Types

A `CHECKBOXES` attribute editor is appropriate for attributes with the following data types:

- date
- float
- integer
- money
- string
- asset (if asset, you must supply the name of the query asset that returns the names of the assets)

## EWEBEDITPRO Example

The `presentationobject.dtd` defines an `EWEBEDITPRO` tag. An attribute editor that uses the tag invokes the `EWEBEDITPRO` element which launches the eWebEditPro HTML editor in a separate window. The person creating the flex asset enters the value for the attribute in that window.

Note the following about creating an eWebEditPro field with an attribute editor:

- You must have the eWebEditPro application installed and configured correctly. It is not delivered with CS-Direct Advantage. For information about configuring eWebEditPro, see the *Content Server Administrator's Guide*.
- It is highly recommended that you use eWebEditPro only when the data type of the attribute is set to `blob`. If you use `blob` as the data type, you do not have to worry about sizing the field.

The `EWEBEDITPRO` tag takes the following parameters:

- `XSIZE`—the x axis dimension, in pixels.
- `YSIZE`— the y axis dimension, in pixels.

The following attribute editor code includes an `EWEBEDIT` pro tag that creates text box that is 400 pixels wide by 200 pixels high:

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT SYSTEM "presentationobject.dtd">

<PRESENTATIONOBJECT NAME="eWebEditProTest">
   <EWEBEDITPRO XSIZE="400" YSIZE="200">
   </EWEBEDITPRO>
</PRESENTATIONOBJECT>
```

### Notes About Data Types

The best choice for the data type of an attribute that uses an `EWEBEDITPRO` attribute editor is `blob`. You can use `string` or `text` but it is problematic because it is hard to predict how large the data entered into the attribute's field will be because each HTML marker counts toward the limit. The `string` data type is limited to 256 characters and `text` is limited to 2000.

FatWire recommends that you use `blob` as the data type for attributes that use eWebEditPro as their input mechanism.

## PICKASSET Example

The `presentationobject.dtd` defines a `PICKASSET` tag—an attribute editor that uses the tag invokes the `PICKASSET` element, which formats a field that accepts the value of an asset in one of two ways, depending on whether the tree is toggled on or off.

- If the tree in the Content Server interface is toggled on, the `PICKASSET` element uses the Pick From Tree method. That is, you select an asset by clicking on it in one of the tabs on the tree in the left frame of the CS-Direct Advantage window.
- If the tree is toggled off, the `PICKASSET` element uses the Remember method, pop-up window that displays all the assets that are currently listed in your Active List and History list.

This tag has no default parameters.

Here is the code to create a `PICKASSET` attribute editor:

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT SYSTEM "presentationobject.dtd">

<PRESENTATIONOBJECT NAME="PickAsset">
   <PICKASSET>
   </PICKASSET>
</PRESENTATIONOBJECT>
```

### Notes About Data Types

A `PICKASSET` attribute editor is only appropriate for attributes with a data type of `asset`.

## PULLDOWN Example

The `presentationobject.dtd` defines a `PULLDOWN` tag—an attribute editor that uses the tag invokes the `PULLDOWN` element, which formats a field with a drop-down list of values.

This tag takes the following parameters:

- `ITEM` or `QUERYASSETNAME` — the source of the names in the drop-down list. To specify the names, use the `ITEM` parameter. To specify a query asset that obtains the names dynamically from a database table, use the `QUERYASSET` parameter.

  Note the following:

  - You cannot use a SQL statement — you must use a query asset if you want to use a query.
  - The SQL in the query asset must return a “value” column. For example: `select name as value from shippingtype`
  - If the data type of the attribute using the attribute editor is “asset”, the query must also return the assets’ IDs. For example: `select name as value, id as assetid from Product where...`

The following attribute editor code specifies that the list holds the items red, green, and blue:

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT SYSTEM "presentationobject.dtd">

<PRESENTATIONOBJECT NAME="PulldownTest">
<PULLDOWN>
```

```
    <ITEM>Red</ITEM>
    <ITEM>Green</ITEM>
    <ITEM>Blue</ITEM>
</PULLDOWN>
```

For example code that shows how to use the QUERYASSETNAME parameter rather than ITEM, see "CHECKBOXES Example" on page 356.

### Notes About Data Types

A PULLDOWN attribute editor is appropriate for attributes with the following data types:

- date
- float
- integer
- money
- string
- asset

A pull-down list is the default input style for attributes of type asset. The list displays all the assets of that type. Use a PULLDOWN attribute editor when you want to further restrict the items in the drop-down list with a query asset so that the list doesn't display every asset of that type.

## RADIOBUTTONS Example

The presentationobject.dtd defines a RADIOBUTTONS tag—an attribute editor that uses the tag invokes the RADIOBUTTONS element, which creates a set of radio options for the attribute.

The RADIOBUTTONS tag takes the following parameters:

- ITEM or QUERYASSETNAME — the source of the names listed next to the radio options. To specify the names, use the ITEM parameter. To specify a query asset that obtains the names dynamically from a database table, use the QUERYASSET parameter.

  Note the following:

  - You cannot use a SQL statement—you must use a query asset if you want to use a query.
  - The SQL in the query asset must return a "value" column. For example: `select name as value from shippingtype`
  - If the data type of the attribute using the attribute editor is "asset", the query must also return the assets' IDs. For example: `select name as value, id as assetid from Product where...`
- LAYOUT— whether the buttons should be positioned in a vertical list or spread out in a horizontal row. Valid options are HORIZONTAL or VERTICAL. The default is HORIZONTAL.

The following attribute editor code specifies that the RADIOBUTTONS element should use the results of a query asset named A Prods for the names of a vertical list of buttons:

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT SYSTEM "presentationobject.dtd">

<PRESENTATIONOBJECT NAME="RadioButtonTest">
```

```
   <RADIOBUTTONS LAYOUT="VERTICAL">
         <QUERYASSETNAME>A Prods</QUERYASSETNAME>
   </RADIOBUTTONS>
</PRESENTATIONOBJECT>
```

For example code that shows the use of the `ITEM` parameter, see "PULLDOWN Example" on page 358.

### Notes About Data Types

A `RADIONBUTTON` attribute editor is appropriate for attributes with the following data types:

- date
- float
- integer
- money
- string
- asset (if asset, you must supply the name of the query asset that returns the names of the assets)

## TEXTAREA Example

The `presentationobject.dtd` defines a `TEXTAREA` tag— an attribute editor that uses the tag invokes the `TEXTAREA` element, which creates a text box field for the attribute, and a pair of radio buttons that allows users to specify whether or not that attribute should display embedded link buttons.

The `TEXTAREA` tag takes the following parameters:

- `XSIZE` — the x axis dimension, in pixels.
- `YSIZE` — the y axis dimension, in pixels.
- `WRAPSTYLE` — whether the text in the box wraps at all, and, if it does whether it wraps automatically (soft) or only when the user presses the Enter key (a hard return). Valid options are `SOFT`, `HARD`, and `OFF`. The default is `OFF`.
- `DEPTYPE` — the approval dependency between the main asset and an embedded asset. Valid options are `EXISTS` and `EXACT`. The default is `EXACT`.
  - `EXISTS` — when the main asset is edited, approved, and re-published, the embedded asset does not need to be approved and re-published as long as a version of the asset already exists at the destination.
  - `EXACT` — when the main asset is edited, approved, and re-published, the embedded asset, if it was edited, must be approved and re-published as well.

The following attribute editor code defines the `XSIZE` as 40 pixels, the `YSIZE` as 5 pixels, disables text wrapping by setting `WRAPSTYLE` to `OFF`, and sets the approval dependency for embedded assets to `EXISTS`:

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT SYSTEM "presentationobject.dtd">

<PRESENTATIONOBJECT NAME="TextAreaTest">
   <TEXTAREA XSIZE="40" YSIZE="5" WRAPSTYLE="OFF"
   DEPTYPE="EXISTS">
```

```
   </TEXTAREA>
</PRESENTATIONOBJECT>
```

### Notes About Data Types

A TEXTAREA attribute editor is appropriate for attributes with the data type of text and blob. Use the text data type when you need to store up to 2000 characters. If you need to store more than 2000 characters, use the blob data type.

## TEXTFIELD Example

The presentationobject.dtd defines a TEXTFIELD tag—an attribute editor that uses the tag invokes the TEXTFIELD element from the New and Edit forms, which creates a text field for the attribute. When the attribute is displayed on the "Inspect" form, however, it uses the DisplayTEXTFIELD element.

The TEXTFIELD tag takes the following parameters:

- XSIZE — the length of the field, in characters.
- MAXCHARS — the number of characters, up to 256, allowed in the field.
- BLANKED — whether the attribute's value is replaced with a string of asterisks when it is displayed in the "Inspect" form. For example, if you created a "password" attribute, you would not want the value of the password displayed in an "Inspect" form. Valid options are YES and NO. The default is NO.

  Because using the BLANKED parameter automatically means that you need the field to behave differently on the "New" and "Edit" forms than it does on the "Inspect" form, the TEXTFIELD tag is delivered with both of the two possible elements by default.

The following attribute editor code defines the XSIZE as 60 and the maximum number of characters as 80:

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT SYSTEM "presentationobject.dtd">

<PRESENTATIONOBJECT NAME="TextFieldTest">
   <TEXTFIELD XSIZE="60" MAXCHARS="80">
   </TEXTFIELD>
</PRESENTATIONOBJECT>
```

### Notes About Data Types

A TEXTFIELD attribute editor is appropriate for attributes with the following data types:

- float
- integer
- money
- string
- url

## The Attribute Editor Elements

The elements that take the input values passed to them from their attribute editor counterparts supply the logic behind the format and behavior of the attribute when it is displayed on a form. For example, it might perform a loop sequence for multivalue attributes so that additional values can be entered in the field.

Following are the default flex attribute display elements located in the `ElementCatalog` table under `OpenMarket/Gator/AttributeTypes`. The names of these elements match exactly the names of the custom XML tags defined in the `presentationobject.dtd` file:

| Element | Description |
|---|---|
| `CHECKBOXES` | Formats the input style of the attribute as a set of check box options. The attribute editor must either define the names of the options or provide the name of a query asset to use to obtain the names. |
| `EWEBEDITPRO` | Invokes the eWebEditPro WYSIWYG editor, a third-party product available from Ektron, Inc.<br>The attribute editor must specify the x and y pixel dimensions. |
| `PULLDOWN` | Formats the input style of the attribute as a select field with a drop-down list. The attribute editor must either specify the items that are displayed in the list or provide the name of a query asset to use to obtain the values. |
| `RADIOBUTTONS` | Formats the input style of the attribute as a set of radio options. The attribute editor must define the names of the options or provide the name of a query asset to use to obtain the names. |
| `TEXTAREA` | Formats the input style of the attribute as a text box and displays radio buttons that allow the user to specify whether or not the text box will allow embedded links. The attribute editor must define the x and y dimensions of the box. |
| `TEXTFIELD` | Formats the input style of the attribute as a text field. The attribute editor must define the length of the field and the number of characters that are allowed in the field. |
| `DisplayTEXTFIELD` | Formats the appearance of the text field attribute's value when it is displayed on the "Inspect" form. If the attribute editor sets the `BLANKED` parameter to `YES`, this element displays the value from the field as a string of asterisks. Typically used for password fields. |
| `PICKASSET` | Formats the input style of the attribute to change based on whether the tree is toggled off or on:<br>• When the tree is displayed, the attribute uses the "pick from tree" mechanism.<br>• When the tree is not displayed, the attribute uses the "remember" mechanism. |
| `PICKFROMTREE` | Deprecated. Use `PICKASSET`. |
| `REMEMBER` | Formats the input style of the attribute as a popup window that displays all the assets that are currently on the user's Active List and History tabs. |

## Conventions for the Attribute Editor Elements

In order for CS-Direct Advantage to use an element for an attribute editor, that element must conform to the following rules:

- It must have the same name as the input style tag that calls it from the attribute editor code. For example, the default `CHECKBOXES` tag has a default `CHECKBOXES.xml` element.
- The element must be placed in the `ElementCatalog` using the following naming conventions: `OpenMarket/Gator/AttributeTypes/`*`name`*

If you want to create your own display elements to use with custom attribute editors, it is best to find one that is the closest to the attribute editor element that you want to create and then copy as much of it as possible.

For help, examine the code in the default attribute editor elements and read the following descriptions of the variables and syntax in them.

### Variables

When CS-Direct Advantage loads a form that uses the attribute editor, it calls the element with the machine name. It passes the information in the following variables to the display element:

- `PresInst`—the instance of the current presentation object
- `AttrName`—the name of the current attribute
- `AttrType`—the data type of the current attribute
- `EditingStyle`—whether the attribute can take more than one value (based on the value in the **Number of Values** field for the attribute). This variable is set to either `single` or `multiple`.
- `RequiredAttr`—whether or not the attribute is required for the current asset. The variable is set to either `true` or `false`.
- `MultiValueEntry`—instructs CS-Direct Advantage how to handle the values for an attribute that can take more than one value.

  When this value is set to `yes`, the display element is called once, under the assumption that the widget created by the element enables the user to select more than one value in it (a multi-select drop-down list, for example).

  When this value is set to `no`, CS-Direct Advantage calls the display element once for each possible value for the attribute and displays one widget for each value that can be stored.

  Note that this value is always set to `yes` initially.
- `doDefaultDisplay`—whether to use the default input style for an attribute of this type. (For a list, see "Default Input Styles for Attributes" on page 199.) When CS-Direct Advantage calls the display element, this variable is initially set to `yes`. To use the input widget created by the element, the element must reset this variable to `no`.
- `AttrValueList`—the list of all the values for this attribute.
- `TempVal`—the value of a single attribute value.

### Other Required Syntax

The code in the display element must also use the following conventions:

- It must store information about how to validate the attribute values in a variable named `RequireInfo`. CS-Direct Advantage passes this variable elements use JavaScript to validate the attribute values. Those elements are:

  ```
  OpenMarket/Gator/FlexibleAssets/FlexAssets/ContentForm1
  OpenMarket/Gator/FlexibleAssets/FlexGroups/ContentForm1
  ```

  This JavaScript performs prescribed error checking and validation based on the type of control, the data type, and other predictable characteristics. The information passed in the `RequireInfo` variable informs the JavaScript about the custom requirements for the attribute editor.
- The name of the widget in the display element (the `INPUT NAME`) must use the following convention:
  - For a single-value attribute, the name of the attribute.
  - For a multi-value attribute, it must use a 1-based counter prepend the attribute name for each attribute value (for example, 1color, 2color, 3color).

For an example, see "Customizing Attribute Editors" on page 367.

# Creating Attribute Editors

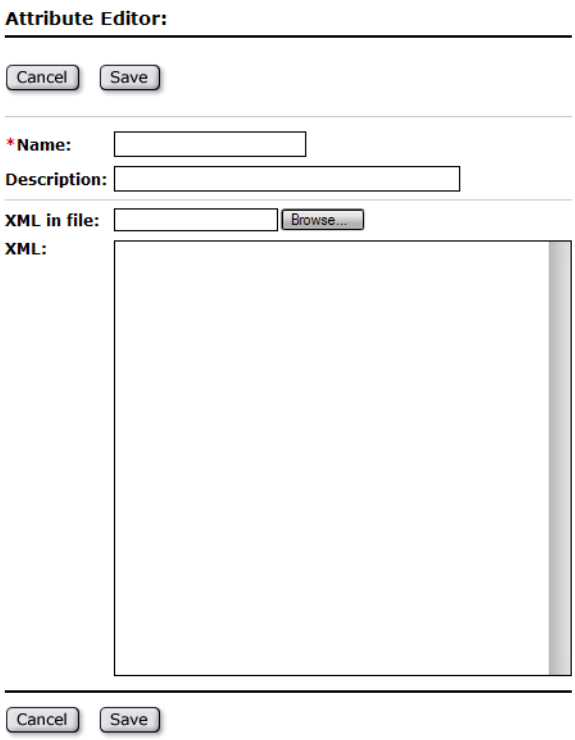
To create an attribute editor using the sample XML code provided in this chapter or in the sample text files, complete the following steps:

1. Open your browser and enter this address:

   `http://your_server/Xcelerate/LoginPage.html`

2. Enter your login name and password and click **Login**.
3. Click **New** and select **Attribute Editor** from the shortcut list

   The **New Attribute Editor** form appears:




4. Click in the **Name** field and enter a unique name of up to 64 characters, excluding spaces.
5. Click in the **Description** field and enter a short phrase that describes the purpose of the attribute editor.
6. Click in the **XML** field. Either cut and paste the appropriate sample XML attribute editor code from the HTML version of this guide or from the sample text files provided in the `Samples` subdirectory of the installation directory.
7. Edit the code as needed. For example, if you are creating a `CHECKBOXES` or a `RADIOBUTTONS` attribute editor, you must provide names for the check boxes or radio buttons. If you are creating a `PULLDOWN` attribute editor, you must provide the values for the drop-down list.

   See "The Attribute Editor Asset" on page 355 for more information about coding the attribute editor.

8. Click **Save**.

> **Note**
>
> Another option is to code the XML for the attribute editor in a separate `.xml` file. In this case, rather than enter the code directly into the XML field, click the **Browse** button next to the XML in file field and select the file.

9. Before this attribute editor can be published to the management system, you must Approve it. For information about approving assets, see the *Content Server Advanced Interface User's Guide*.

> **Note**
>
> If you are using a query asset with this attribute editor, be sure to approve both the attribute editor and the query asset.
>
> Because the dependency between an attribute editor and its query asset is specified in the XML code in the attribute editor, the approval system can not detect the dependency and verify that the query asset exists on the management system.

.

# Customizing Attribute Editors

If you need to create your own custom attribute editor, the best thing to do is to copy as much as you can from the sample attribute editor code and the sample display elements.

If you determine that you must create a new input style (you cannot use any of the default `PRESENTATIONOBJECT` tags), you must add a new `PRESENTATIONOBJECT` section that to the `presentationobject.dtd` file that defines the attribute editor. For information about adding to this file, see "The presentationobject.dtd File" on page 353.

When you create a custom `PRESENTATIONOBJECT` tag, you must also supply the appropriate display elements for it:

- Required: An element that formats the attribute (displays an edit mechanism) when that attribute appears in a "New" or "Edit" form.
- Optional: An element that formats the attribute when it appears in the "Inspect" form.
- Optional: An element that formats the attribute data before it is saved in the database.

For information about the variables and conventions used in the display elements for an attribute editor, see "The Attribute Editor Elements" on page 361.

## Example: Customized Attribute Editor

This example demonstrates how you could customize the description of the `TEXTAREA` tag in the `presentationobject.dtd` file and the `TEXTAREA` element to create an attribute editor that disables a text box if the user does not have the proper permissions.

There are three steps:

1. Editing the description of the `TEXTAREA` tag in the `presentationobject.dtd` to support a new parameter named `PERMISSIONS`.
2. Writing the code for the attribute editor and creating the attribute editor.
3. Editing the `TEXTAREA` element to check the value of `PERMISSIONS`.

### Step 1: Editing the presentationobject.dtd file

To support the new parameter, you add a single line of code to the `TEXTAREA` description in the `presentationobject.dtd`:

```
1   <!-- TEXTAREA: A text area of a specific size. You must
    specify
2   -- the x and y dimensions; the wrap style defaults to soft.
3   -->
4   <!ELEMENT TEXTAREA ANY>
5   <!ATTLIST TEXTAREA XSIZE CDATA #REQUIRED>
6   <!ATTLIST TEXTAREA YSIZE CDATA #REQUIRED>
7   <!ATTLIST TEXTAREA WRAPSTYLE (OFF | SOFT | HARD) "SOFT">
8   <!ATTLIST TEXTAREA PERMISSION CDATA>
```

The new line of code is line 8. Lines 1 through 7 are the default description of the `TEXTAREA` tag.

## Step 2: Example Code for the Example Attribute Editor

Here's the example code with the new parameter. It specifies that a user must have "Administrators" as the value for `PERMISSION` in order to see the field:

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT SYSTEM "presentationobject.dtd">

<PRESENTATIONOBJECT NAME="TextAreaTest">
   <TEXTAREA XSIZE="40" YSIZE="10" WRAPSTYLE="SOFT"
         PERMISSION="Administrators">
   </TEXTAREA>
</PRESENTATIONOBJECT>
```

## Step 3: Editing the TEXTAREA Element

The third step is editing the `TEXTAREA` element. Lines 56–70, 123–125, and 172–174 are the new code that enables or disables the field, based on the value of the `PERMISSION` parameter:

```
1   <?XML VERSION="1.0" ?>
2   <!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
3   <FTCS Version="1.1">
4   <!-- OpenMarket/Gator/AttributeTypes/TEXTAREA
5   --
6   -- INPUT
7   --
8   -- OUTPUT
9   --
10  -->
11
12  <!-- Display one TEXTAREA per attribute value -->
13  <IF COND="Variables.MultiValueEntry=no">
14  <THEN>
15
16  <!-- Don't want default display field  -->
17  <setvar NAME="doDefaultDisplay" VALUE="no"/>
18
19  <!-- Get all parameters from Attribute Editor xml -->
20  <presentation.getprimaryattributevalue
21  NAME="Variables.PresInst"
22  ATTRIBUTE="FONTSIZE" VARNAME="FONTSIZE"/>
23     <if COND="Variables.errno!=0">
24  <then>
25  <setvar NAME="FONTSIZE" VALUE="2"/>
26  </then>
27  </if>
28
29  <presentation.getprimaryattributevalue
30  NAME="Variables.PresInst"
31  ATTRIBUTE="WRAPSTYLE" VARNAME="WRAPSTYLE"/>
32  <if COND="IsVariable.WRAPSTYLE!=true">
33  <then>
34  <setvar NAME="WRAPSTYLE" VALUE="OFF"/>
```

```
35 </then>
36 </if>
37
38 <presentation.getprimaryattributevalue
39 NAME="Variables.PresInst"
40 ATTRIBUTE="XSIZE" VARNAME="XSIZE"/>
41 <if COND="IsVariable.XSIZE!=true">
42 <then>
43 <setvar NAME="XSIZE" VALUE="24"/>
44 </then>
45 </if>
46
47 <presentation.getprimaryattributevalue
48 NAME="Variables.PresInst"
49 ATTRIBUTE="YSIZE" VARNAME="YSIZE"/>
50 <if COND="IsVariable.YSIZE!=true">
51 <then>
52 <setvar NAME="YSIZE" VALUE="20"/>
53 </then>
54 </if>
55
56 <setvar NAME="disableTextArea" VALUE="no"/>
57 <presentation.getprimaryattributevalue
58 NAME="Variables.PresInst"
59 ATTRIBUTE="PERMISSION" VARNAME="PERMISSION"/>
60 <if COND="IsVariable.PERMISSION=true">
61 <then>
62 <setvar NAME="errno" VALUE="0"/>
63 <USERISMEMBER GROUP="Variables.PERMISSION"/>
64 <IF COND="Variables.errno=0">
65 <THEN>
66 <setvar NAME="disableTextArea" VALUE="yes"/>
67 </THEN>
68 </IF>
69 </then>
70 </if>
71
72 <tr>
73
74 <!-- Standard element to display attribute name or
   description
75 -->
76 <callelement NAME="OpenMarket/Gator/FlexibleAssets/Common
77 /DisplayAttributeName"/>
78 <td></td>
79 <td>
80
81 <!-- Single valued attributes -->
82 <if COND="Variables.EditingStyle=single">
83 <then>
84
85 <!-- Special case: TEXTAREA for URL attributes -->
```

```
86 <IF COND="Variables.AttrType=url">
87 <THEN>
88 <setvar NAME="errno" VALUE="0"/>
89 <BEGINS STR="AttrValueList.urlvalue"
90 WHAT="AttrValueList."/>
91 <IF COND="Variables.errno=1">
92 <THEN>
93 <setvar NAME="filename" VALUE="CS.UniqueID.txt"/>
94 </THEN>
95 <ELSE>
96 <setvar NAME="filename"
97 VALUE="AttrValueList.urlvalue"/>
98 </ELSE>
99 </IF>
100
101 <INPUT TYPE="hidden" NAME="Variables.AttrName_file"
102 VALUE="Variables.filename"
103 REPLACEALL="Variables.AttrName,Variables.filename"/>
104
105 <setvar NAME="errno" VALUE="0"/>
106 <BEGINS STR="AttrValueList.@urlvalue"
107 WHAT="AttrValueList."/>
108 <IF COND="Variables.errno=1">
109 <THEN>
110 <setvar NAME="MyAttrVal" VALUE="Variables.empty"/>
111 </THEN>
112 <ELSE>
113 <setvar NAME="MyAttrVal"
114 VALUE="AttrValueList.@urlvalue"/>
115 </ELSE>
116 </IF>
117 </THEN>
118 </IF>
119
120 <!-- Display a TEXTAREA with all parameters from Attribute
121 --Editor xml -->
122 <!-- The NAME of the input must be the attribute name -->
123 <IF COND="Variables.disableTextArea=yes">
124 <THEN>
125 <TEXTAREA DISABLED="yes" NAME="Variables.AttrName"
126 ROWS="Variables.YSIZE" COLS="Variables.XSIZE"
127 WRAP="Variables.WRAPSTYLE"
128 REPLACEALL="Variables.AttrName,Variables.XSIZE,
129 Variables.YSIZE,Variables.WRAPSTYLE,Variables.empty">
130
131 <!-- For most single valued attrs, the value is contained
    in
132 MyAttrVal -->
133 <csvar NAME="Variables.MyAttrVal"/>
134 </TEXTAREA>
135 </THEN>
136 <ELSE>
```

```
137 <TEXTAREA NAME="Variables.AttrName"
138 ROWS="Variables.YSIZE" COLS="Variables.XSIZE"
139 WRAP="Variables.WRAPSTYLE"
140 REPLACEALL="Variables.AttrName,Variables.XSIZE,
141 Variables.YSIZE,Variables.WRAPSTYLE,Variables.empty">
142 <!-- For most single valued attrs, the value is
143 contained in MyAttrVal -->
144 <csvar NAME="Variables.MyAttrVal"/>
145 </TEXTAREA>
146 </ELSE>
147 </IF>
148 </then>
149 <else>
150 <!-- Multiple valued attributes -->
151 <!-- For single value attributes we can usually use the
152 default RequireInfo -->
153 <!-- For multiple value attributes we need to append to
154 RequireInfo for each value -->
155 <if COND="Variables.RequiredAttr=true">
156 <then>
157 <setvar NAME="RequireInfo"
158 VALUE="Variables.RequireInfo*Counters.TCounterVariables.
159 AttrName*ReqTrue*Variables.AttrType!"/>
160 </then>
161 <else>
162 <setvar NAME="RequireInfo"
163 VALUE="Variables.RequireInfo*Counters.TCounterVariables
164 .AttrName*ReqFalse*Variables.AttrType!"/>
165 </else>
166 </if>
167
168 <!-- Display a TEXTAREA with all parameters from Attribute
169 Editor xml -->
170 <!-- The NAME of the input must be the attribute name
171 prepended by the TCounter counter -->
172 <IF COND="Variables.disableTextArea=yes">
173 <THEN>
174 <TEXTAREA DISABLED ="yes"
    NAME="Counters.TCounterVariables.AttrName"
175 ROWS="Variables.YSIZE" COLS="Variables.XSIZE"
176 WRAP="Variables.WRAPSTYLE"
177 REPLACEALL="Counters.TCounter,
178 Variables.AttrName,Variables.XSIZE,
179 Variables.YSIZE,Variables.WRAPSTYLE">
180 <csvar NAME="Variables.tempval"/> </TEXTAREA>
181 </THEN>
182 <ELSE>
183 <TEXTAREA NAME="Counters.TCounterVariables.AttrName"
184 ROWS="Variables.YSIZE" COLS="Variables.XSIZE"
185 WRAP="Variables.WRAPSTYLE"
186 REPLACEALL="Counters.TCounter,
187 Variables.AttrName,Variables.XSIZE,
```

```
188 Variables.YSIZE,Variables.WRAPSTYLE">
189 <csvar NAME="Variables.tempval"/> </TEXTAREA>
190 </ELSE>
191 </IF>
192 </else>
193 </if>
194 </td>
195 </tr>
196 </THEN>
197 </IF> <!-- MultiValueEntry -->
198 </FTCS>
```

# Editing Attribute Editors

Note the following when editing an attribute editor:

- You can change the **Name** without causing a schema change.
- You can change the **Description** without causing data loss.
- If you change code in the attribute editor:
  - You can add input options.
  - If you have existing data, you should not remove input options. If you do, some of your existing data will no longer be valid and you will have to search through the database and fix it.
  - If you change the input style, you risk a data mismatch.

Chapter 19

# Configuring Bundled Attribute Editors

This chapter explains how to configure instances of attribute editors that ship with Content Server.

This chapter contains the following sections:


- Configuring FCKEditor
- Configuring the Clarkii Online Image Editor
- Configuring the Image Picker
- Configuring the RENDERFLASH Attribute Editor

# Configuring FCKEditor

FCKEditor is a third-party HTML editor from Frederico Caldeira Knabben that ships bundled with Content Server. In the FirstSite II sample site, selected asset types containing WYSIWYG-enabled text fields are configured to use FCKEditor by default.

This section covers the following topics:


- Before You Begin
- Creating an FCKEditor Instance and Enabling It for a Field
- Enabling FCKEditor for Use in the InSite Interface
- Implementing Lazyload
- Enabling Selected Asset Types for the FCKEditor
- Enable "Image Picker" for the FCKEditor
- Customizing the FCKEditor Toolbar
- Configuring Spell Check Support in FCKEditor


## Before You Begin

If you are using FCKEditor with Internet Explorer, and Content Server is configured to use the cp1252 character set, complete the following steps to have FCKEditor work correctly in Internet Explorer:

1. Open the `futuretense.ini` property file and note the value of the `cs.contenttype` property.
2. Log in to Content Server Explorer.
3. Go to the element `SiteCatalog\OpenMarket\xcelerate\Actions\FCKEditorRenderer` and set its `resargs1` field to:

   ```
   cs.contenttype=[your content type]; charset=windows-1252
   ```

   where `[your content type]` takes the value that is set in the `futuretense.ini` file.

## Creating an FCKEditor Instance and Enabling It for a Field

This procedure shows you how to create an instance of FCKEditor and enable it for a flex asset attribute based on a FirstSite II example.

In our example, you will enable a new FCKEditor instance as the input method (attribute editor) for the **FSIIAbstract** field. (**FSIIAbstract** is a flex attribute of the "Content" sample asset type.)

### To create and enable an FCKEditor instance

1. Log in to the Advanced interface as an administrator (`fwadmin/xceladmin` in our example).
2. Select the site in which you want to work.
3. Create the FCKEditor instance:
   a. In the button bar, click **New**.

b. In the list of asset types, click **New Attribute Editor**.

   Content Server displays the "New Attribute Editor" form.

c. In the **Name** field, enter a name that uniquely identifies this instance of FCKEditor. For the purpose of our example, enter **FCK_FSIIAbstract**.

d. Paste the following code into the **XML** field:

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT SYSTEM
   "presentationobject.dtd">
<PRESENTATIONOBJECT NAME="FCK_FSIIAbstract">
   <FCKEDITOR XSIZE="400" YSIZE="200"></FCKEDITOR>
</PRESENTATIONOBJECT>
```

e. Click **Save**.

4. Enable the FCKEditor instance as the input method for the **FSIIAbstract** field:

   a. Find and open the **FSIIAbstract** attribute asset in the "Edit" form:

      1) In the button bar, click **Search**.

      2) In the list of asset types, click **Find Content Attribute**.

      3) In the search field, enter **FSIIAbstract**.

      4) Click **Search**.

      5) In the list of search results, navigate to the **FSIIAbstract** asset and click its **Edit** (pencil) icon.

         Content Server opens the asset in the "Edit" form.

   b. Set FCKEditor as the attribute editor (input method) for this attribute. In the "Attribute Editor" drop-down list, select **FCK_FSIIAbstract**.

   c. Click **Save** to save your changes.

5. Test your new FCKEditor instance:

   a. Find and open any "Content" asset in the "Edit" form:

      1) In the button bar, click **Search**.

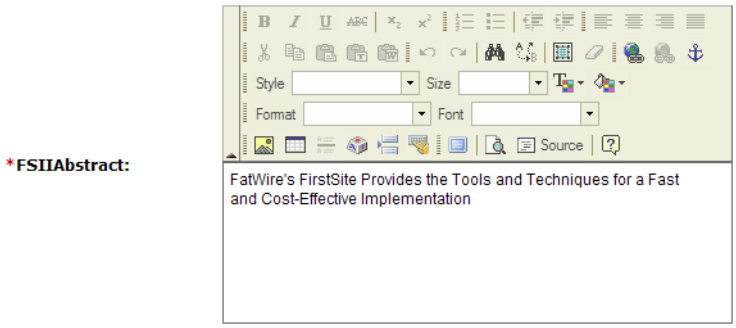
      2) In the list of asset types, click **Find Content**.

      3) Click **Search**.

      4) In the list of asset types, navigate to a "Content" asset of your choice (**FSIIAbout** in our example) and click its **Edit** (pencil) icon.

         Content Server displays the asset in the "Edit" form.

**b.** Navigate to the **FSIIAbstract** field. It should look as follows:




If FCKEditor does not appear, check the attribute editor XML code and the selection you made in the "Attribute Editor" drop-down list you made in step 4b above.

**c.** Click **Cancel** to return to the asset's "Inspect" screen.

## Enabling FCKEditor for Use in the InSite Interface

To enable users to edit assets using FCKEditor in the InSite interface, you will need to set the following parameters – `editor` and `params` – in the `insite:edit` tag within the appropriate template. (The template whose code you edit must be either for the asset whose FCKEditor you wish to work with in the InSite interface, or for the asset type associated with that asset.)

**Table 10:** Parameters that Enable FCKEditor in the InSite Interface

| Tag | Parameter | Value | Description |
|---|---|---|---|
| `insite:edit` | `editor` | `fckeditor` | Specifies the name of the editor to use. |
| | `params` | `editorId` | Specifies the ID of the FCKEditor you want to use in the InSite interface. |
| | | `enableEmbeddedLinks` | Enables the link icons on the FCKEditor toolbar. |

```
<ics:listget listname='BodyList' fieldname="value" output="Body" />
<div id="body">
<insite:edit
      assetid='<%=ics.GetVar("cid")%>'
      assetfield='<%="Attribute_"+ics.GetVar("BodyAttrName")%>'
      assetfieldvalue='<%=ics.GetVar("Body")%>'
      assettype='<%=ics.GetVar("c")%>'
      editor='fckeditor'
      params-"{params:{editorId: '1170996054393',enableEmbeddedLinks:'1'}}"/>
</div>
```

Parameters required to enable FCKEditor in the InSite interface.

**Note**

If the `insite:edit` tag does not define a height and width for FCKEditor, then FCKEditor will use the `XSIZE` and `YSIZE` defined in its XML code. For more information see the *Content Server Tag Reference*.

You can edit the `insite:edit` tag in one of the following:

- In the "Element Logic" field of the appropriate template. To work in a template's "Element Logic" field:
  1. Log in to the Advanced interface as an administrator and select the desired site.
  2. Click **Search** (in the button bar) **> Find Template > Search >** *Select template name.*
  3. From the template's "Edit" form, select the **Element** tab and navigate to the "Element Logic field."
- CS-Explorer (for instructions on editing code with Content Server Explorer, see "Using Content Server Explorer to Create and Edit Element Logic," on page 540).
- A text editor of your choice.

## Implementing Lazyload

When many FCKEditors are configured for a given asset (for example, an FCKEditor for each of five fields), the process of loading the asset's "New" or "Edit" form can take a long time. To minimize the time it takes to load an asset form, you have the option to implement a feature called "lazyload." When you add the `LAZYLOAD` parameter to the FCKEditor's attribute editor, and set the `LAZYLOAD` parameter to `true`, FCKEditors in a given asset will not automatically load with the asset form. Because less elements are loaded at the same time as the asset form, the loading process is faster.

To work in an FCKEditor-enabled field when `LAZYLOAD` is implemented, the user must initiate FCKEditor for that field by clicking in the field. When the user clicks in the field, the FCKEditor toolbar opens.

**Note**

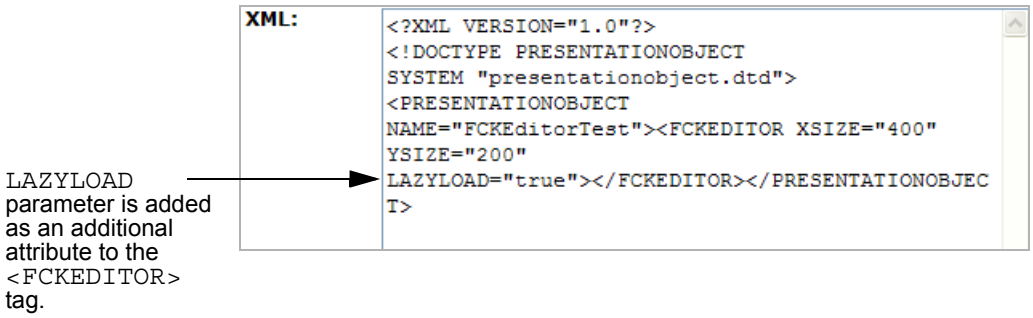
By default, the value of the `lazyload` parameter is set to `false` in the FCKEditor for the Advanced interface, and set to `true` in the FCKEditor for the Dash interface.

#### To implement and test the lazyload parameter

1. Log in to Content Server's Advanced interface as an administrator.
2. Select the site in which you want to work.
3. Find and open the attribute editor named FCKEditor in its "Edit "form:
   a. In the button bar, click **Search**.
   b. In the "Search" form select **Find Attribute Editor**.
   c. In the "Search for Attribute Editors" form, fill in the desired search criteria (if any) and click **Search**.

   d. In the search results list navigate to the **FCKEditor** asset and click its **Edit** (pencil) icon.

4. Navigate to the XML field, add the `LAZYLOAD` parameter to the `<FCKEDITOR>` tag, and set the value of the `LAZYLOAD` parameter to either `true` or `false`.




5. Click **Save Changes** to save the asset.
6. Test the `LAZYLOAD` parameter:
   a. Open the "Edit" form of the asset you will use to test the `LAZYLOAD` parameter. The asset you choose must have an FCKEditor enabled field.
   b. Click in the field to make sure the FCKEditor opens.

## Enabling Selected Asset Types for the FCKEditor

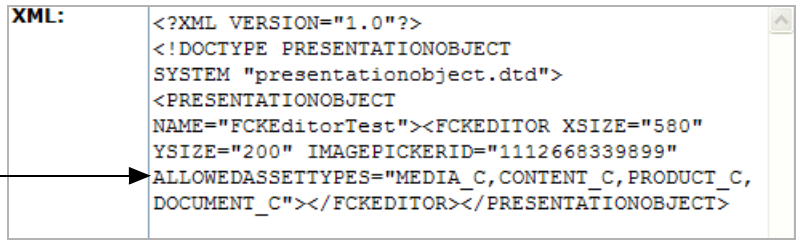
Users can create an asset directly from another asset's FCKEditor enabled field by clicking the **Create and link a new asset** icon or the **Create and include a new asset** icon. By default, all asset types are available for users to create an asset from. To narrow down the user's choice of available asset types, add the `ALLOWEDASSETTYPES` parameter to the FCKEditor XML code. As values, specify the names of the allowed asset types in a comma-separated list.

When a user clicks the appropriate icon in the FCKEditor toolbar, the "Add Asset" window opens, and the asset type(s) you enabled with the `ALLOWEDASSETTYPES` parameter are displayed in the window.

### To enable asset types for the FCKEditor

1. Log in to Content Server's Advanced interface as an administrator.
2. Select the site in which you want to work.
3. Find and open the attribute editor named FCKEditor in its "Edit" form:
   a. In the button bar, click **Search**.
   b. In the "Search" form select **Find Attribute Editor**.
   c. In the "Search for Attribute Editors" form, fill in the desired search criteria (if any) and click **Search**.
   d. In the search results list navigate to the **FCKEditor** asset and click its **Edit** (pencil) icon.

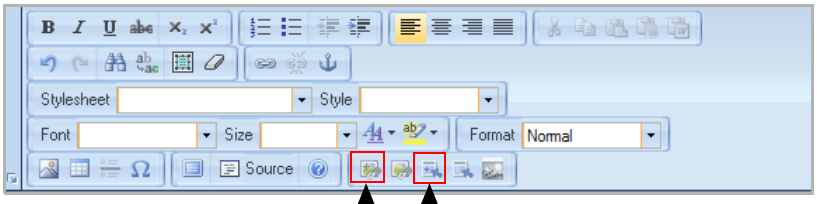
4. Navigate to the XML field, add the `ALLOWEDASSETTYPES` parameter to the `<FCKEDITOR>` tag. Then, specify the name(s) of the asset type(s) you want to enable in the value of the `ALLOWEDASSETTYPES` parameter.




`ALLOWEDASSETTYPES` parameter is added as an additional parameter to the `<FCKEDITOR>` tag.

The value of this parameter specifies the asset types which the user can create a new asset with.

5. Click **Save Changes** to save the asset.
6. Test the `ALLOWEDASSETTYPES` parameter.
   a. Open the "Edit" form of the asset you will use to test the `ALLOWEDASSETTYPES` parameter. The asset you choose must have an FCKEditor enabled field.
   b. In the FCKEditor toolbar, click either the **Create and link a new asset** icon or the **Create and include a new asset** icon.




**Create and link a new asset** icon

**Create and include a new asset** icon

The "Add Asset" window opens and the name(s) of the asset type(s) you specified in the value of the `ALLOWEDASSETTYPES` parameter should be displayed.

## Setting the Approval Dependency for Included Assets

The **Include asset** and **Create and include a new asset** icons allow users to include one asset in another asset's FCKEditor-enabled field. The included asset is then previewable in the field and, ultimately, embedded in the display of the main asset online.

After the main and included assets have been published for the first time, the dependency between them determines how subsequent approvals and publications will work. This dependency is defined by the `DEPTYPE` parameter in the XML code of the FCKEditor.

The `DEPTYPE` parameter can be set to either `EXISTS` or `EXACT`. If the `DEPTYPE` parameter is not set explicitly, `EXACT` is used by default.

- `EXISTS` — When the main asset is edited, approved, and re-published, the included asset does not need to be approved and re-published as long as a version of the asset already exists at the destination.
- `EXACT` — When the main asset is edited, approved, and re-published, the included asset, if it was edited, must be approved and re-published as well.

### To define the approval dependency for included assets

1. Log in to Content Server's Advanced interface as an administrator.
2. Select the site in which you want to work.
3. Find and open the attribute editor named FCKEditor in its "Edit" form:
   a. In the button bar, click **Search**.
   b. In the "Search" form, select **Find Attribute Editor**.
   c. In the "Search for Attribute Editors" form, fill in the desired search criteria (if any) and click **Search**.
   d. In the search results list navigate to the **FCKEditor** asset and click its **Edit** (pencil) icon.
4. Navigate to the "XML" field, add the `DEPTYPE` parameter to the `<FCKEDITOR>` tag, and set the value of the `DEPTYPE` parameter to either `EXISTS` or `EXACT`.

**XML:**

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT
SYSTEM "presentationobject.dtd">
<PRESENTATIONOBJECT
NAME="FCKEditorTest"><FCKEDITOR XSIZE="580"
YSIZE="200"
DEPTYPE="EXISTS"></FCKEDITOR></PRESENTATIONOBJE
CT>
```

`DEPTYPE` parameter is added as an additional attribute to the `<FCKEDITOR>` tag.

5. Click **Save Changes** to save the asset.

## Enable "Image Picker" for the FCKEditor

Users might want to invoke an "Image Picker" from an FCKEditor enabled field of an asset. On the FCKEditor toolbar, the user can click the **Pick an image to include** icon to invoke an "Image Picker." However, by default the icon is disabled. To enable this icon, add the `IMAGEPICKERID` parameter to the FCKEditor's XML code, and set its value to the ID of the "Image Picker" you want to be invoked when a user clicks the **Pick an image to include** icon.

### To enable an "Image Picker" from FCKEditor

1. Log in to Content Server's Advanced interface as an administrator.
2. Select the site in which you want to work.
3. Find and open the attribute editor named FCKEditor in its "Edit" form:
   a. In the button bar, click **Search**.
   b. In the "Search" form, select **Find Attribute Editor**.
   c. In the "Search for Attribute Editors" form, fill in the desired search criteria (if any) and click **Search**.
   d. In the search results list navigate to the **FCKEditor** asset and click its **Edit** (pencil) icon.

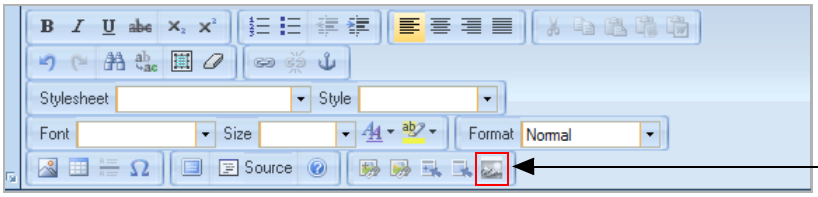
4. Navigate to the XML field, add the `IMAGEPICKERID` parameter to the `<FCKEDITOR>` tag, set the value to the ID of the "Image Picker" you want to be invoked when the **Pick an image to include** icon is clicked.

```
XML:
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT
SYSTEM "presentationobject.dtd">
<PRESENTATIONOBJECT
NAME="FCKEditorTest"><FCKEDITOR XSIZE="580"
YSIZE="200" IMAGEPICKERID="1112668339899"
ALLOWEDASSETTYPES="MEDIA_C,CONTENT_C,PRODUCT_C,
DOCUMENT_C"></FCKEDITOR></PRESENTATIONOBJECT>
```

`IMAGEPICKERID` parameter is added as an additional attribute to the `<FCKEDITOR>` tag.

The value of this parameter is the ID of the "Image Picker" that will be invoked from the FCKEditor.

5. Click **Save Changes** to save the asset.
6. Test the `IMAGEPICKERID` parameter:
   a. Open the "Edit" form of the asset you will use to test the `IMAGEPICKERID` parameter. The asset you choose must have an FCKEditor enabled field.
   b. In the FCKEditor toolbar, click the **Pick an image to include** icon.




**Pick an image to include** icon

   The "Image Picker," whose ID you set as the value for the `IMAGEPICKERID` parameter, opens and the image assets associated with that "Image Picker" are rendered in the window.

## Customizing the FCKEditor Toolbar

You have the ability to customize the functions available in FCKEditor's toolbar, as well as their arrangement. You can customize the FCKEditor toolbar either system-wide or on a per-instance basis.

### Customizing the FCKEditor Toolbar System-Wide

1. Open the following file in a text editor:

   `<cs_app>/js/fckconfig.js`

   where `<cs_app>` refers to the directory into which the CS web application was deployed on your application server.
2. Locate the `FCKConfig.ToolbarSets["CS"]` toolbar definition. This toolbar definition determines the contents of the FCKEditor toolbar system-wide.
3. Make your changes. For information on the toolbar definition syntax, consult the FCKEditor documentation.
4. Save and close the file.

5. Redeploy the CS application and restart the application server for your changes to take effect.

## Customizing the FCKEditor Toolbar on a Per-Instance Basis

1. Create the custom toolbar definition:
   a. Open the following file in a text editor:

      `<cs_app>/js/fckconfig.js`

      where `<cs_app>` refers to the directory into which the CS web application was deployed on your application server.

   b. Add a new toolbar definition at the end of the file. For information on how to build a custom toolbar definition, consult the FCKEditor documentation.

      ```
      FCKConfig.ToolbarSets["<toolbarDef>"] = [
      ['Bold','Italic','Underline'],'/',
      ['Cut','Copy','Paste'],'/',
      ['FitWindow','-','Preview','Source','-','About']
      ];
      ```

      The `<toolbarDef>` value is the name of your custom toolbar definition. You will use this name in step 2 on page 382 when modifying the desired FCKEditor instance to use this custom definition.

2. Modify the desired FCKEditor instance to use your custom toolbar definition.
   a. Log in to Content Server's Advanced interface as an administrator.
   b. Select the site in which you want to work.
   c. Find the desired FCKEditor instance:
      1) In the button bar, click **Search**.
      2) In the list of asset types, click **Find Attribute Editor.**
      3) In the **Search** field, enter the name of the asset holding the desired FCKEditor instance and click **Search**.
      4) In the list of search results, navigate to the desired asset and click its **Edit** (pencil) icon.
   d. In the **XML** field, modify the attribute editor code as follows:
      1) Find the line highlighted in bold below:

      ```
      <?XML VERSION="1.0"?>
      <!DOCTYPE PRESENTATIONOBJECT SYSTEM
         "presentationobject.dtd">
      <PRESENTATIONOBJECT NAME="FCKEditorCustomized">
      <FCKEDITOR XSIZE="400" YSIZE="200">
      </FCKEDITOR></PRESENTATIONOBJECT>
      ```

      2) Add the `TOOLBAR` parameter to the end of the line, but before the closing `>` bracket:

      **`TOOLBAR="<toolbarDef>"`**

      The value of the `TOOLBAR` parameter specifies the name of the custom toolbar definition you created in step 1 on page 382.

The modified line should look as follows:

```
<FCKEDITOR XSIZE="400" YSIZE="200" TOOLBAR="<toolbarDef>">
```

e. Click **Save Changes** to save the asset.

3. Redeploy the CS application and restart the application server for your changes to take effect.

For more information on configuring FCKEditor, consult its documentation.

## Configuring Spell Check Support in FCKEditor

FCKEditor supports integration with the following spell check software:

- ieSpell – a client-side spell checker plugin for Microsoft Internet Explorer
- Spell Pages – a server-side open-source spell checker application

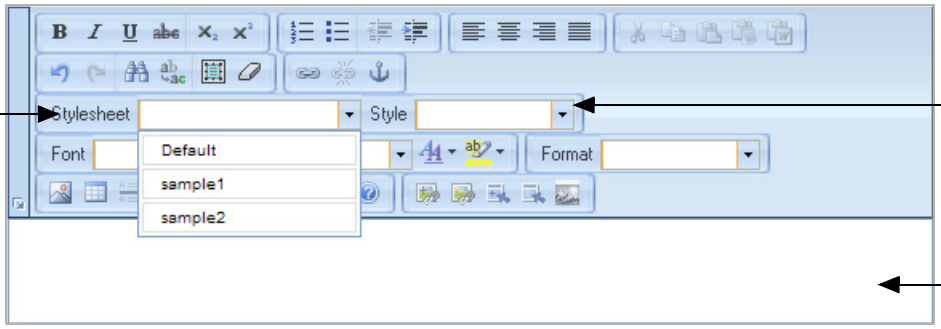
For instructions on configuring FCKEditor to support these spell checkers, visit the following URL:

`http://wiki.fckeditor.net/Developer's_Guide/Configuration/Spell_Checker`

## Enabling Custom Style Sheets for FCKEditor

This section shows you how to configure the style sheet plugin in order to add custom XML style sheets to the “Stylesheet” menu on the FCKEditor toolbar. This plugin applies to all FCKEditor enabled fields, system-wide.

When the plugin is configured and the “Stylesheet” menu is opened, the style sheets that you have specified are listed in the menu. Once a user selects a style sheet, the styles defined by that style sheet are listed in the “Style” menu, on the FCKEditor toolbar. The user can then apply these style options to format text in the editor area of the FCKEditor.




This menu lists custom XML style sheets, along with the default style sheet.

This menu lists the styles defined by the style sheet you selected from the “Stylesheet” menu.

Editor area.

If a selected style sheet has an associated CSS, the CSS file loads into the editor area automatically, but without explicitly displaying.

The style sheet plugin also enables you to associate custom style sheets with CSS files, by coding them appropriately. When a style sheet is selected from the “Stylesheet” menu, the associated CSS loads in the editor area of the FCKEditor. To learn more about XML styles, see `http://docs.fckeditor.net/FCKeditor_2.x/Developers_Guide/Configuration/Styles`.

**To enable custom style sheets for the FCKEditor**

1. Configure the style sheet plugin:

   In order for custom XML style sheet files to be listed in the "Stylesheet" menu on the FCKEditor toolbar, you must set the values of the following configuration properties in the `fckconfig.js` file (located in `WebApp/cs/js`):

   **Table 11:** FCKEditor Properties and Sample Values

| Property | Description |
|---|---|
| `BaseCustomStylePath` | This property points to the directory containing all the XML files that have been created, when the context root of the directory is added to the value of the property. (This directory can contain any number of XML style sheets.)<br>**Example:**<br>`FCKConfig.BaseCustomStylePath = 'FCKeditor/editor/css/' ;` |
| `AllXMLStylesheets` | This property is used to specify which of the XML style sheets, found in the `BASECUSTOMSTYLEPATH` directory, are going to be listed in the "Stylesheet" menu on the FCKEditor toolbar.<br>**Example:**<br>`FCKConfig.AllXMLStylesheets = 'sample1;sample2';` |
| `BaseCustomCSSPath` | This property points to the directory containing all CSS files, when the context root of the directory is added to the value of the property. For example, if `/cs` is the context root, then the FCKEditor will be the directory for the `cs directory` (`cs.war`) in the application server.<br>**Example:**<br>`FCKConfig.BaseCustomCSSPath = 'FCKeditor/editor/css/' ;` |

2. If you need to create the XML style sheet(s), you can create a stand-alone style sheet (as many as necessary), or you can associate the style sheet(s) with a CSS file. Procedures begin in the next section.

## Creating and Testing Style Sheets

This section starts with the simpler case of creating a stand-alone XML style sheet (`My XML.xml`). If you wish to skip this procedure and create an XML style sheet that is associated with a CSS file, see "Creating and Testing an XML Style Sheet That Uses a CSS File," on page 388.

### Creating and Testing a Stand-Alone XML Style Sheet

An XML style sheet is considered stand-alone when all attribute values for a given style are specified in the style sheet (rather than a CSS).

1. Create an XML style sheet. The name can contain spaces, underscores and any combination of alphanumeric characters. The name will be displayed in the "Stylesheet" menu on the FCKEditor toolbar exactly as you write it, but without the extension.

   In this example we create a file named `My XML.xml`.

   **My XML.xml**

```
      <Style name="Heading 1" element="div">
          <Style name="font-size" value="large" />
         <Style name="font-weight" value="bold" />
         <Style name="color" value="#036" />
         <Style name="border-bottom" value="1px solid #036" />
      </Style>
         <Style name="Block Quote" element="div">
          <Style name="margin" value="0 10px" />
          <Style name="padding" value="5px" />
         <Style name="border" value="1px solid #99c" />
         <Style name="background-color" value="#ccf" />
         <Style name="color" value="#036"/>
      </Style>
      <Style name="No Wrap" element="span">
          <Style name="white-space" value="nowrap" />
      </Style>
</Styles>
```

2. Enable the style sheet which you created, so that it will be listed in the **Stylesheet** menu of the FCKEditor, and the specified styles will be listed in the **Style** menu:

   **a.** Place the style sheet in the directory specified by the `BaseCustomStylePath` property.

   **b.** Specify the name of the style sheet as a value of the `AllXMLStylesheets` property.

   **Note**

   The properties are described in Table 11, on page 384.

3. Test the stand-alone XML style sheet.

   **a.** Open the "Edit" form of the asset you will use to test the style sheet you created. The asset you choose must have a field that is FCKEditor enabled.

   If `LAZYLOAD` has been implemented, you must initiate the FCKEditor for the field by clicking in the field (the FCKEditor toolbar then opens).

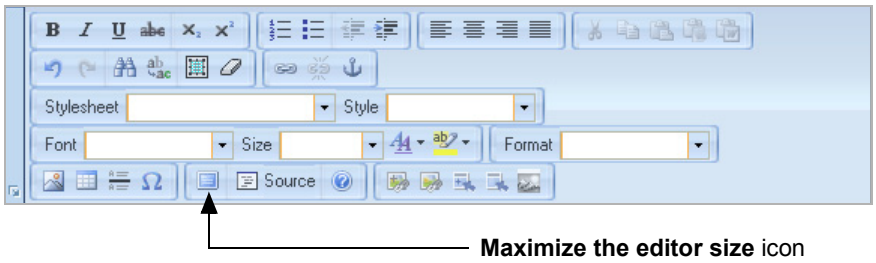
   **b.** On the FCKEditor toolbar, make sure:

   **1)** The style sheet that you created (`MY XML` in this example) is listed in the "Stylesheet" menu.

   **2)** The styles that you specified in the style sheet are listed in the "Style" menu.

   **c.** Apply the styles to ensure they format the text appropriately:

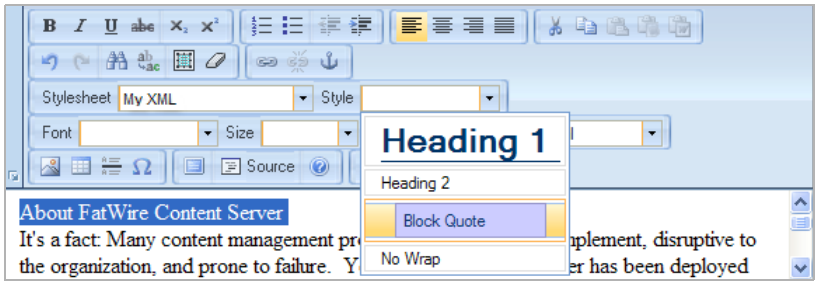
1) Expand the editor area (click the **Maximize the editor size** icon on the FCKEditor toolbar).



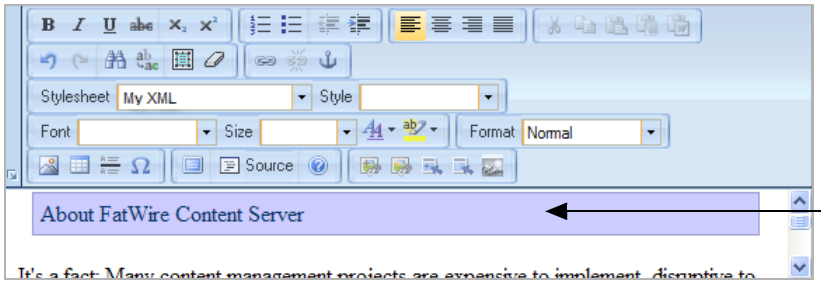

2) Select the text you wish to format.

3) Go to the "Stylesheet" menu and select the style sheet you created (**MY XML** in this example).

4) From the "Style" menu, select a style (**Block Quote** in this example).

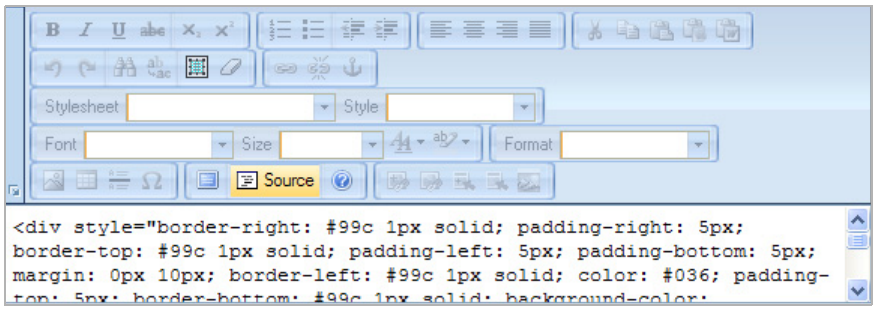



The text is now formatted with the style you selected.




Text formatted with **Block Quote** style.

**5)** Check the coding of the text to which you applied the style (click the **Source** option on the FCKEditor toolbar to view the HTML code).




**Note**

In your style sheet, you may have included `<div>` and `<span>` elements. If so, apply the styles as follows:

- Styles with a `<div>` element
  - To apply the style to a single paragraph, click anywhere in the paragraph and select the style.
  - To apply the style to a block of paragraphs, select the block and apply the style.
- The `<span>` element applies to selected text. Select all the text that you wish to format and apply the style.

## Creating and Testing an XML Style Sheet That Uses a CSS File

If necessary, create a CSS file and a corresponding XML style sheet. You can modify an existing style sheet so it refers to the CSS file, or you can create a new XML style sheet.

**Note**

The FCKEditor can load only one CSS at a time in the editor area. Therefore, an XML style sheet can be associated with only one CSS.

1. Create a CSS file (`XML_CSS.css` in this example). The name can contain spaces, underscores, and any combination of alphanumeric characters, as long as the name of the CSS file is identical to the associated XML file.

   **XML_CSS.css**

   ```
   div.heading2{
            font-size : medium;
            font-weight : bold;
            font-style : italic;
            color : #99c;
         }
   ```

2. Create the XML style sheet and give it the name of the CSS file, (`XML_CSS.xml` in this example). The name can contain spaces, underscores and any combination of alphanumeric characters. The name of the style sheet will be displayed in the "Stylesheet" menu exactly how you write it, but without the extension.

   **Note**

   The style sheet and its associated CSS file must have the same file name. Identical names ensure that the correct CSS file is referred to by the style sheet.

   In this example, we copied the `MY XML.xml` file (page 385), and gave it the name of the CSS file (`XML_CSS.xml` in this example). Then, we added the "`Heading 2`" style name (in the CSS) to `XML_CSS.xml` (lines 7 and 8).

   **XML_CSS.xml**

   ```
   1  <Style name="Heading 1" element="div">
   2  <Style name="font-size" value="large" />
   3  <Style name="font-weight" value="bold" />
   4  <Style name="color" value="#036" />
   5  <Style name="border-bottom" value="1px solid #036" />
   6  </Style>
   7  <Style name="Heading 2" element="div">
   8  <Attribute name="class" value="heading2" />
   9  </Style>
   10 <Style name="Block Quote" element="div">
   11     <Style name="margin" value="0 10px" />
   12     <Style name="padding" value="5px" />
   13 <Style name="border" value="1px solid #99c" />
   14 <Style name="background-color" value="#ccf" />
   ```

```
15 <Style name="color" value="#036"/>
16 </Style>
17 <Style name="No Wrap" element="span">
18     <Style name="white-space" value="nowrap" />
19 </Style>
20 </Styles>
```

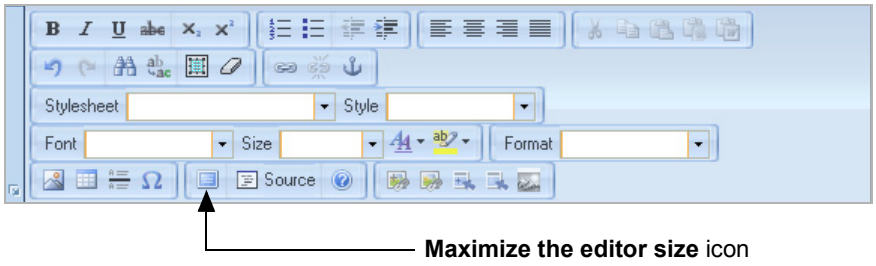
3. Enable the style sheet and CSS which you created so the style sheet is displayed in the **Stylesheet** menu of the FCKEditor, and the specified styles are displayed in the **Style** menu:
   a. Place the style sheet into the directory that is specified by the `BaseCustomStylePath` property.
   b. Specify the name of the style sheet as a value of the `AllXMLStylesheets` property.
   c. Place the CSS file into the directory that is specified by the `BaseCustomCSSPath` property.

**Note**

The properties are described in Table 11, on page 384.

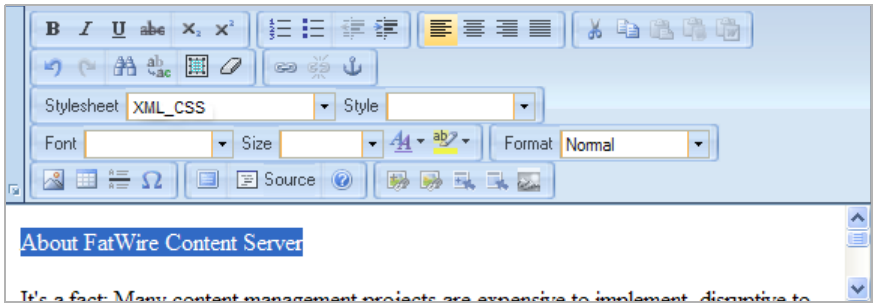
4. Test the style sheet and the corresponding CSS file.
   a. Open the "Edit" form of the asset you will use to test the style sheet and CSS file. The asset you choose must have an FCKEditor enabled field.

      If `LAZYLOAD` has been implemented, you must initiate the FCKEditor for that field by clicking in it (the FCKEditor toolbar then opens).
   b. On the FCKEditor toolbar, make sure:
      1) The style sheet that you created (`XML_CSS` in this example) is listed in the "Stylesheet" menu.
      2) The styles defined in the style sheet and CSS are listed in the "Style" menu. (The association between a style and a CSS is not explicitly shown in any of the menus.)
   c. Apply the styles to ensure they format the text appropriately:
      1) Expand the editor area (click the **Maximize the editor size** icon on the FCKEditor toolbar).

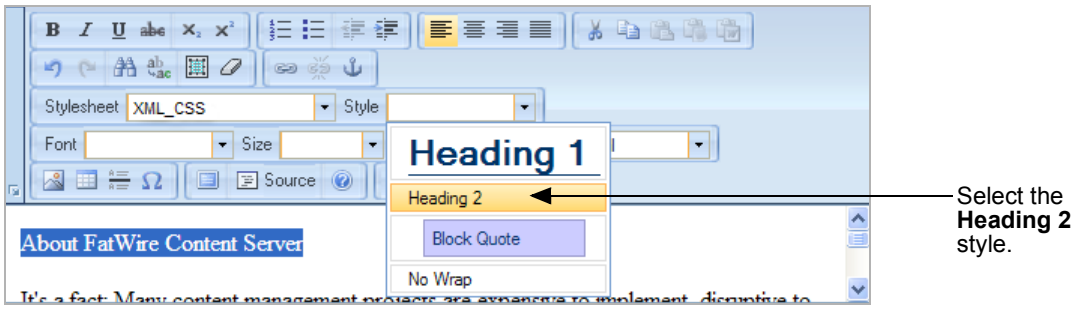



      2) From the "Stylesheet" menu, select the style sheet you created (**XML_CSS** in this example).

**3)** Select the text to which you will apply the style.



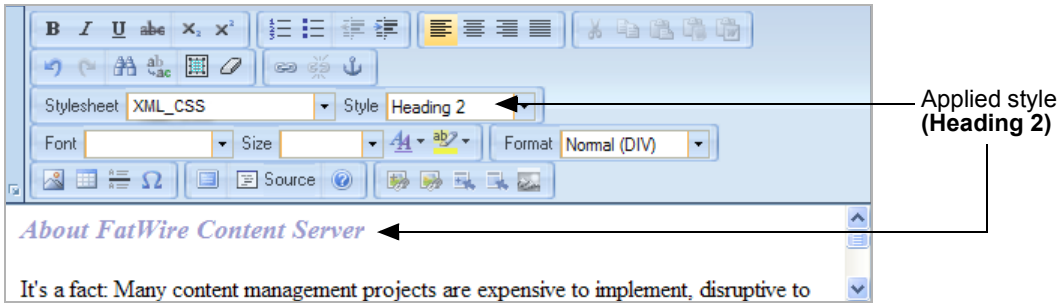

**4)** From the "Style" menu, select a style (**Heading 2** in this example).



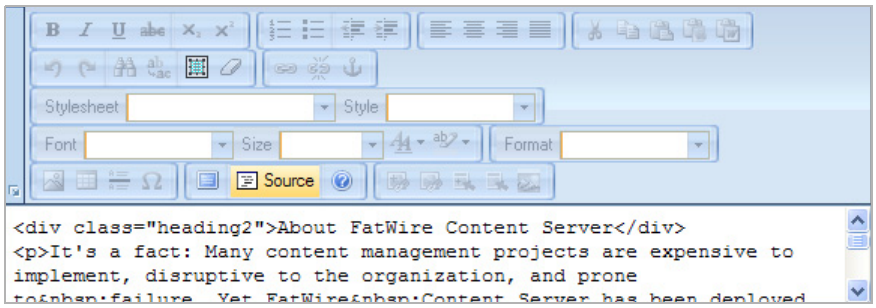

**Note**

When you select a style whose attribute values are defined by the CSS (Heading 2 in this example), the CSS loads automatically into the editor area of the FCKEditor.

The text should be formatted with the style you selected.



**5)** Check the coding of the formatted text (select the **Source** option to view the HTML code).




In this example, the selected text "About FatWire Content Server" has been replaced by: `<div class="heading2">About FatWire Content Server</div>`

**Note**

In your style sheet and/or CSS file, you may have included `<div>` and `<span>` elements. If so, apply the styles as follows:

- Styles with a `<div>` element:
  - To apply the style to a single paragraph, click anywhere in the paragraph and select the style.
  - To apply the style to a block of paragraphs, select the block and apply the style.
- The `<span>` element applies to selected text. Select all the text that you wish to format and apply the style.

If you now choose a different style sheet, the HTML code remains in the editor area, but the formatting applied by the previous style sheet and CSS is lost, unless the styles in the new style sheet and CSS match those of the previous style sheet and CSS.

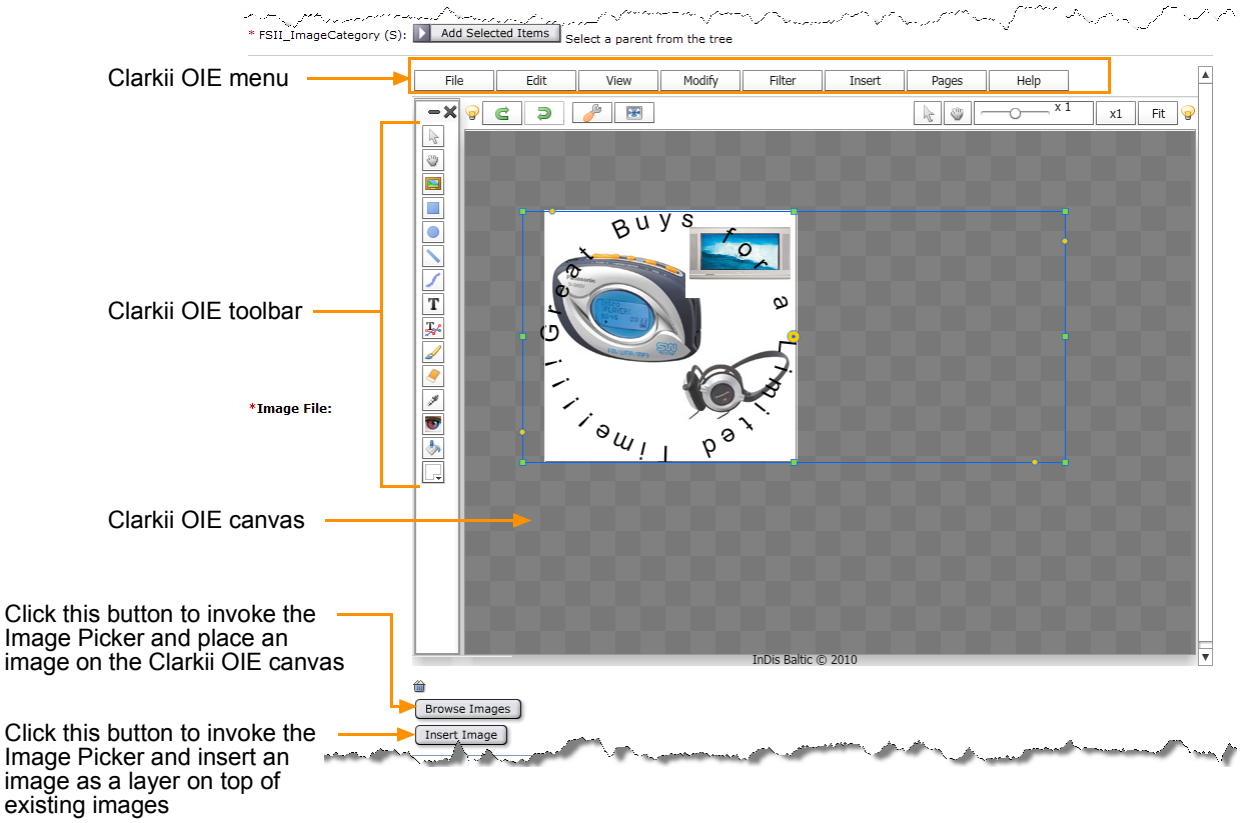
# Configuring the Clarkii Online Image Editor

The Clarkii Online Image Editor (Clarkii OIE) is a third-party image editor from InDis Baltic that is supported on all browsers on which Content Server is supported, including Safari.

If you will be using the FirstSite II sample site to create a Clarkii OIE instance, you can modify the code for the Online Image Editor (OIE) that ships bundled with Content Server by adding the tag `EDITORTYPE="clarkii"`. If you are using a site other than the FirstSite II sample site, you can either configure separate instances of Clarkii OIE on a per-field basis for each asset type, or you can configure a single instance of Clarkii OIE to be associated with the fields of multiple asset types.

Figure 6 summarizes the native controls of Clarkii OIE and functions provided by Content Server for operating on images in an attribute for which Clarkii OIE is enabled.

**Figure 6:** Clarkii Online Image Editor rendered in a field of an asset's form



Before configuring and enabling this feature, take note of the following:

- Clarkii OIE can be enabled only for flex attributes. The instructions in this section use the "Media" flex family of the FirstSite II sample site as an example.
- Clarkii OIE can be enabled only for attributes of type `blob`.
- Flash must be installed on the client browser in order for Clarkii OIE to be rendered in the field for which you enabled it.

**Note**

You can customize the functions in the Clarkii OIE toolbar and menu. Since these functions are strictly Clarkii OIE related they are not documented. For instructions about configuring Clarkii OIE specific functions, visit the following URL:

`http://www.online-image-editor-clarkii.com/`

This section contains the following topics:

- Creating a Clarkii OIE Instance and Enabling it for a Field
- Configuring Clarkii OIE Properties
- Enabling Clarkii OIE for Use in the InSite Interface

## Creating a Clarkii OIE Instance and Enabling it for a Field

This procedure shows you how to create a Clarkii OIE instance and enable it for a flex attribute asset. This procedure is based on the FirstSite II sample site.

In this example, you will enable a new Clarkii OIE instance as the attribute editor for the **FSII_ImageFile** field. **FSII_ImageFile** is an attribute of the sample "Media" asset type.

1. Log in to the Advanced interface as a general administrator (`fwadmin`/`xceladmin` in this example).
2. Create a Clarkii OIE instance:
   a. In the button bar, click **New**.
   b. In the list of asset types, click **New Attribute Editor**.

      Content Server displays the "New Attribute Editor" form.
   c. In the **Name** field, enter a name that uniquely identifies this Clarkii OIE instance (for this example, enter **ClarkiiOIE**).
   d. Paste the following code into the **XML** field.

      **Note**

      For detailed information about each parameter, see the table in "Configuring Clarkii OIE Properties," on page 396, or click a parameter in the code below to view its description and possible values.

```
1 <?XML VERSION="1.0"?>
2 <!DOCTYPE PRESENTATIONOBJECT SYSTEM
   "presentationobject.dtd">
3 <PRESENTATIONOBJECT NAME="editor">
```

```
4 <IMAGEEDITOR
5      HEIGHT="600"
6      WIDTH="800"
7      EDITORTYPE="clarkii"
8      FITIMAGE="false"
9      ENABLEIMAGEPICKER="true"
10     ASSETTYPE="Media_C"
11     ATTRIBUTE="FSII_ImageFile"
12     ATTRIBUTETYPE="Media_A"
13     CATEGORYATTRIBUTE=""
14     RESTRICTEDCATEGORYLIST=""
15     OIEENABLEIMAGEPICKER="true"
16     OIEASSETTYPE="Media_C"
17     OIEATTRIBUTE="FSII_ImageFile"
18     OIEATTRIBUTETYPE="Media_A"
19     OIECATEGORYATTRIBUTE=""
20     OIERESTRICTEDCATEGORYLIST="">
21 </IMAGEEDITOR>
22 </PRESENTATIONOBJECT>
```

**e.** Click **Save**.

**3.** Enable the Clarkii OIE instance as the attribute editor for an attribute of a given asset type (for this example, use the “Media” asset type’s **FSII_ImageFile** attribute).

**a.** Find and open the desired attribute asset in its “Edit” form (for this example, find and open the **FSII_ImageFile** attribute).

**1)** In the button bar, click **Search**.

**2)** In the list of asset types, click the asset type of the desired attribute asset (for this example, click **Find Media Attribute**).

**3)** In the search field, enter the name of the attribute asset you wish to modify (for this example, enter **FSII_ImageFile**).

**4)** Click **Search**.

**5)** In the list of search results, navigate to the desired attribute asset (**FSII_ImageFile**) and click its **Edit** (pencil) icon.

Content Server opens the asset in its “Edit” form.

**b.** Set Clarkii OIE as the attribute editor for this attribute asset:

**1)** In the “Value Type” field, make sure `blob` is selected.

> **Note**
>
> Clarkii OIE requires an attribute value of type `blob`. Once an attribute asset is saved, the value selected for the “Value Type” field cannot be modified.

**2)** In the “Attribute Editor” drop-down list, select the Clarkii OIE instance you created in step 2 on page 393.

**c.** Click **Save Changes**.

**4.** Test your new Clarkii OIE instance:

**a.** Find any asset whose definition specifies the attribute you modified in step 3 as a field, and open that asset in its “Edit” form (for this example, select a “Media” asset):

**1)** In the button bar, click **Search**.

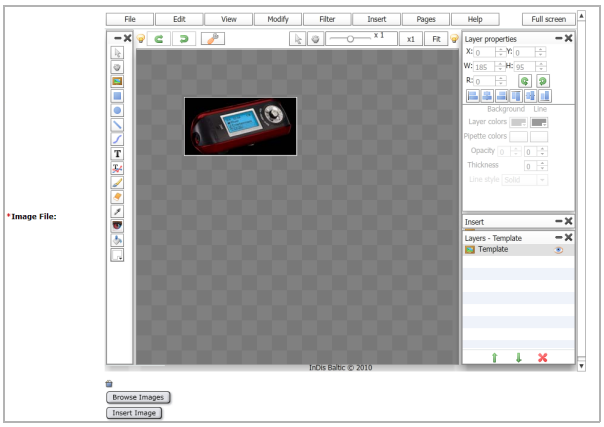
**2)** In the list of asset types, click the asset type using the field enabled with the Clarkii OIE instance (for this example, select **Find Media**).

**3)** Click **Search**.

**4)** In the list of search results, navigate to an asset of your choice (for this example, select **FSII AudioCo_iAC-800.jpg** “Media” asset) and click its **Edit** (pencil) icon.

Content Server displays the asset in its “Edit” form.

**b.** Navigate to the field enabled with Clarkii OIE (in this example, **FSII_ImageFile** field). It should look as follows:




If Clarkii OIE is not rendered in the field, do one of the following:

- Make sure Flash is installed.
- Check the XML code of the Clarkii OIE instance you created (see, step d on page 393).
- Check the selections you made in the attribute asset for which you enabled Clarkii OIE as the attribute editor (see step b on page 394).

**c.** Click **Cancel** to return to the asset’s “Inspect” screen.

## Configuring Clarkii OIE Properties

The table in this section lists and defines all of the properties that can be specified in the creation of a Clarkii OIE attribute editor asset. Use this table as a reference to configure the properties for your own Clarkii OIE instance so it fits your site design.

**Note**

If you wish to view a property within the context of a sample XML code for Clarkii OIE, click the name of the property in the "Property" column of the table.

**Table 12:** Clarkii OIE Properties

| Property | Definition |
|---|---|
| **Clarkii OIE Specifications** | |
| `HEIGHT` | Specify the height of the Clarkii OIE area as it will be displayed within the attribute field of a given asset's form.<br>**Suggested value:** `600` |
| `WIDTH` | Specify the width of the Clarkii OIE area as it will be displayed within the attribute field of a given asset's form.<br>**Suggested value:** `800` |
| `EDITORTYPE` | Specify the type of image editor you wish to use. To enable Clarkii OIE, the property must read:<br>`EDITORTYPE="clarkii"` |
| `FITIMAGE` | If this property is set to `true`, when an image is edited with Clarkii OIE, that image will be resized to fit within the Clarkii OIE canvas. If this property is set to `false`, then when you edit an image with Clarkii OIE, that image is displayed on the canvas in its actual size.<br>**Possible values:** `true` \| `false` |
| **'Browse Images' Button Specifications** | |
| `ENABLEIMAGEPICKER` | Enables the **Browse Images** button which allows users to place an image on the Clarkii OIE canvas, replacing any images that currently exist on the canvas. This button invokes the Image Picker window.<br>**Possible values:** `true`\|`false`<br>**Note:** If this property is set to `false`, do not set values for the other properties related to the **Browse Images** button. |

**Table 12:** Clarkii OIE Properties (continued)

| Property | Definition |
|---|---|
| ASSETTYPE | Specify the asset type of the image assets that will be displayed in the Image Picker window when the **Browse Images** button is clicked.<br>**Possible values:** Any asset type that has a definition containing an attribute of type `blob` intended to store images.<br>**Example:** `"Media_C"` |
| ATTRIBUTE | Specify the image file attribute of the image assets that will be displayed in the Image Picker window when the **Browse Images** button is clicked.<br>**Possible values:** Any attribute of type `blob` intended to store images.<br>**Example:** `"FSII_ImageFile"` |
| ATTRIBUTETYPE | Specify the asset type of the image file attribute specified in the `ATTRIBUTE` property. Image Picker will look for attributes of only this asset type, and displays only the images with attributes of this asset type.<br>**Possible values:** The asset type of the image file attribute specified in the `ATTRIBUTE` property.<br>**Example:** `"Media_A"` |
| CATEGORYATTRIBUTE | Specify a `string` attribute holding a value that classifies the image assets (preferably by the names of their parent assets) to be displayed in the Image Picker window. The value of this property populates the "Category" drop-down list in the Image Picker with all values found for the specified attribute. For information about how a field copier filter can be used to classify assets, see "Implementing a Field Copier Filter to Classify Assets," on page 399.<br>When the **Browse Images** button is clicked, users can use the "Category" drop-down list to filter the images that are displayed in the Image Picker window by selecting one of these attribute values. Only the images matching the selected attribute value are rendered in the Image Picker window.<br>**Note:** If no value is specified for this property, all image assets of the asset type specified in the `ASSETTYPE` property are displayed in the Image Picker window, and the "Category" drop-down list is not displayed. |

**Table 12:** Clarkii OIE Properties (continued)

| Property | Definition |
|---|---|
| RESTRICTEDCATEGORYLIST | In a comma-delimited list, enter specific values of the attribute specified in the CATEGORYATTRIBUTE property. The values you specify will be the only values available in the "Category" drop-down list of the Image Picker window when the **Browse Images** button is clicked.<br>**Possible values:** Refer to the definition for the CATEGORYATTRIBUTE property.<br>**Note:** If no value is specified for the CATEGORYATTRIBUTE property, then this property is ignored. |
| **'Insert Image' Button Specifications** | |
| OIEENABLEIMAGEPICKER | Enables the **Insert Image** button, which enables users to insert an image as a layer on top of existing images on the Clarkii OIE canvas. This button invokes the Image Picker window.<br>**Possible values:** true\|false<br>**Note:** If this property is set to false, do not set values for the other properties related to the **Insert Image** button. |
| OIEASSETTYPE | Specify the asset type of the image assets that will be displayed in the Image Picker window when the **Insert Image** button is clicked.<br>**Possible values:** Any asset type that has a definition containing an attribute of type blob intended to store images.<br>**Example:** "Media_C" |
| OIEATTRIBUTE | Specify the image file attribute of the image assets that will be displayed in the Image Picker window when the **Insert Image** button is clicked.<br>**Possible values:** Any attribute of type blob intended to store images.<br>**Example:** "FSII_ImageFile" |
| OIEATTRIBUTETYPE | Specify the asset type of the image file attribute you specified in the OIEATTRIBUTE property. Image Picker will look for attributes of only this asset type, and displays only the images with attributes of this asset type.<br>**Possible values:** The asset type of the image file attribute specified in the OIEATTRIBUTE property.<br>**Example:** "Media_A" |

**Table 12:** Clarkii OIE Properties (continued)

| Property | Definition |
|---|---|
| OIECATEGORYATTRIBUTE | Specify a string attribute holding a value that classifies the image assets (preferably by the names of their parent assets) that will be displayed in the Image Picker window. The value of this property populates the "Category" drop-down list in the Image Picker with all values found for the specified attribute. For information about how a field copier filter can be used to classify assets, see "Implementing a Field Copier Filter to Classify Assets," on page 399.<br>When the **Insert Image** button is clicked, users can use the "Category" drop-down list to filter the images that are displayed in the Image Picker window by selecting one of these attribute values. Only the images matching the selected attribute value are rendered in the Image Picker window.<br>**Note:** If no value is specified for this property, all image assets of the asset type specified in the OIEASSETTYPE property are displayed in the Image Picker window, and the "Category" drop-down list is not displayed. |
| OIERESTRICTEDCATEGORYLIST | In a comma-delimited list, enter specific values of the attribute specified in the OIECATEGORYATTRIBUTE property. The values you specify will be the only values available in the "Category" drop-down list of the Image Picker window when the **Insert Image** button is clicked.<br>**Possible values:** Refer to the definition for the OIECATEGORYATTRIBUTE property.<br>**Note:** If no value is specified for the OIECATEGORYATTRIBUTE property, then this property is ignored. |

## Implementing a Field Copier Filter to Classify Assets

If you wish to classify image assets by the names of their parent assets, you can use a field copier filter. Figure 7 illustrates an example of how to implement a field copier filter, using the "Media" flex family of the FirstSite II sample site. The field copier filter in this example copies the value of the system-defined "Name" attribute of the parent assets into a user defined string attribute.

To ensure that the values of preexisting attributes are not overwritten, you can create a new string attribute (**FSII_ImageParentName**) to hold the value of the system-defined attribute to be copied by the field copier.

**Note**

For more information about Field Copier filters, see "Field Copier Filter Class," on page 336.

**Figure 7:** Overview of using a field copier filter (using the "Media" flex family of the FirstSite II sample site as an example)

**1.** Create the attribute **FSII_ImageParentName** (type `string`), which will be referenced in the field copier filter (step 2).

**2.** Create a filter of type `FieldCopier`. In the "Arguments" field, name the user-defined attribute **FSII_ImageParentName**. In step 4, the filter will copy the value of the system-defined **Name** attribute of the "Media" parent assets into the user-defined attribute.

**User-Defined Attribute**

Media Attribute: FSII_ImageParentName

Inspect | Edit | Delete | more... | Add to My Active List

| | |
|---|---|
| Name: | FSII_ImageParentName |
| Description: | FSII_ImageParentName |
| Status: | Created |
| ID: | 1283892669217 |
| Site: | FirstSite II |
| Value Type: | string |
| Number of Values: | single |
| Attribute Editor: | |
| Created: | Wednesday, September 8, 2010 6:09:01 PM EDT by fwadmin |
| Modified: | Wednesday, September 8, 2010 6:09:01 PM EDT by fwadmin |

**Field Copier Filter**

Media Filter: FSII_ImageParentNameCopier

Inspect | Edit | Delete | more... | Add to My Active List

| | |
|---|---|
| Name: | FSII_ImageParentNameCopier |
| Description: | FSII_ImageParentNameCopier |
| Status: | Created |
| ID: | 1283892669232 |
| Site: | FirstSite II |
| Filter: | FieldCopier |
| Arguments: | name=FSII_ImageParentName |
| Created: | Wednesday, September 8, 2010 6:09:34 PM EDT by fwadmin |
| Modified: | Wednesday, September 8, 2010 6:09:34 PM EDT by fwadmin |

**3.** Add the field copier filter to the desired parent definition. In this example, the filter is added to the **FSII_ImageCategory** "Media" parent definition.

**Note:** Adding the filter rather than the user-defined attribute to the parent definition enables field copying to occur transparently, i.e., the attribute will not be displayed in the parent's "Edit" screen when the parent is saved. Instead, the attribute will be displayed in the parent's "Inspect" screen (step 4).

**Parent Definition**

Inspect Media Parent Definition: FSII_ImageCategory

Inspect | Edit | Delete | more... | Add to My Active List

| | |
|---|---|
| Name: | FSII_ImageCategory |
| Description: | Image Category |
| Status: | Edited |
| ID: | 1112195133969 |
| Parent Select Style: | Pick from Tree |
| Parent Definitions: | (none) |
| Attribute Names: | (none) |
| Filters: | FSII_ImageParentNameCopier |
| Created: | Tuesday, October 19, 2004 1:27:33 PM EDT by admin |
| Modified: | Wednesday, September 8, 2010 6:09:45 PM EDT by fwadmin |

**4.** When a parent using this definition is saved, the field copier filter copies the value of the system-defined **Name** attribute into the user-defined **FSII_ImageParentName** attribute.

**Parent**

Media Parent ( FSII_ImageCategory): FSII Product Images

Preview | Inspect | Edit | Delete | more... | Add to My Active List

| | |
|---|---|
| Name: | FSII Product Images |
| Description: | Product Images |
| Status: | Edited |
| Start Date: | |
| End Date: | |
| Locale: | |
| ID: | 1112195134086 |
| Select a Template: | No template specified |
| Filename: | |
| Path: | |
| Parent Definition: | FSII_ImageCategory |
| FSII_ImageParentName: | FSII Product Images |

**Child**

Media (FSII_Image): FSII AudioCo_iAC-008.jpg

Preview | Inspect | Edit | Delete | more... | Add to My Active List

| | |
|---|---|
| Name: | FSII AudioCo_iAC-008.jpg |
| Description: | |
| Select a Template: | /FSIIDetail |
| ID: | 1114083738551 |
| External Item ID: | |
| Media Definition: | FSII_Image |
| Parents: | FSII Product Images |
| *Image File: | AudioCo_iAC-008.jpg<br>Dimensions: 185 x 95<br>Size: 3.27 KB<br>Type: image/jpeg |

**5.** Each child of the parent asset transparently inherits the value that is copied to the user-defined **FSII_ImageParentName** attribute. You can now use this value to classify a child asset by its parent.

### To use a field copier filter

1. Log in to the Advanced interface as a general administrator (`fwadmin/xceladmin` in this example)
2. Determine the system-defined attribute whose value you wish to use as the input for the field copier filter, and the user defined `string` attribute to which the field copier will be copying the system-defined attribute's value.

> **Note**
>
> If you do not wish to overwrite the values of preexisting attributes, create a flex attribute of type `string` (**FSII_ImageParentName** in this example). For instructions on creating a new flex attribute, see "Step 4: Create Flex Attributes," on page 317.

3. Create a filter of type `FieldCopier` (**FSII_ImageParentNameCopier** in this example):
   a. In the button bar, click **New**.
   b. In the list of asset types, click **New Media Filter**.

      The "New Media Filter" form is displayed.
   c. In the "Name" field, enter a unique name for this filter (in this example, we use **FSII_ImageParentNameCopier**).
   d. In the "Filter" field, select `FieldCopier` from the drop-down list. Then click **Get Arguments**:
      - In the "Name" drop-down menu, select the system-defined attribute whose value you wish to use as the input value for the field copier filter (**name** in this example).
      - In the "Value" field, type the name of the user defined attribute (**FSII_ImageParentName** in this example) into which the field copier filter will copy the value of the system-defined attribute you specified.
   e. Click **Add** to add the argument to the filter.
   f. Click **Save** to save the filter.
4. Find the desired parent (or child) definition and add the new field copier filter to it (in this example we are adding the field copier filter to the "Media" parent definition **FSII_ImageCategory**):
   a. In the button bar, click **Search**.
   b. In the list of asset types, select **Find Media Parent Definition**.
   c. In the search field, enter the name of the parent definition to which you wish to add the field copier filter (**FSII_ImageCategory**).
   d. Click **Search**.
   e. In the list of search results, navigate to the desired parent (or child) definition and click its **Edit** (pencil) icon.
   f. In the "Edit" form of the parent (or child) definition, navigate to the **Filters** section and select the field copier filter you created in step 3.
   g. Click **Save Changes**.

5. Find and re-save all preexisting parent (or child) assets associated with the definition to which you added the field copier filter (**FSII_ImageParentNameCopier**). This enables the filter to populate the user defined attribute (**FSII_ImageParentName**) with the value of the system-defined attribute (**Name**).

   If you added the filter to a parent definition, all children of the associated parent assets internally inherit the value that the field copier filter copied into the user defined attribute (**FSII_ImageParentName** in this example).

## Enabling Clarkii OIE for Use in the InSite Interface

To allow users to edit image assets using Clarkii OIE in the InSite interface, add the `insite:editasset` tag to your template code. This tag renders an **Edit Asset** button when the InSite interface is in the "Editing" mode. Clicking the button will display the image asset's "Edit" form in a pop-up window, in which users will be able to edit the image in the asset field using Clarkii OIE. For information on the parameters taken by this tag, see the *Content Server Developer's Tag Reference*.

**Note**

If you want the **Edit Asset** button to display a label other than "Edit Asset," specify a value for the optional parameter `displayname` in your `insite:editasset` tag call. If you omit the `displayname` parameter, the button will display the default label **Edit Asset**.

# Configuring the Image Picker

This section lists the parameters whose values you must specify in the XML definition when configuring an instance of the Image Picker attribute editor:

| Parameter | Explanation |
|---|---|
| `ASSETTYPENAME` | Asset type of the image assets that this instance of Image Picker will display.<br>Example: `Media_C` |
| `ATTRIBUTETYPENAME` | Asset type of the image file attribute within the selected image asset type.<br>Example: `Media_A` |
| `ATTRIBUTENAME` | Name of the image file attribute within the selected image asset type.<br>Example: `FSII_ImageFile` |
| `CATEGORYATTRIBUTENAME` | (Optional) Name of the category attribute within the selected image asset type.<br>Example: `FSII_ImageCategory` |
| `RESTRICTEDCATEGORYLIST` | Accepts a comma-delimited list of values of the category attribute within the selected image asset type. The values in this list will appear in the "Category" drop-down list in the Image Picker window. If this parameter is omitted, Image Picker will display all assets belonging to the selected image asset type.<br>Example: `Audio,Video,Photo` |

Sample XML code for an Image Picker definition is included in "Sample Image Picker Attribute Editor Definition Code," on page 404. For instructions on creating new attribute editors (such as new instances of Image Picker) see "Creating Attribute Editors," on page 365. For instructions on selecting an input method (such as Image Picker) for a field in an asset form, see "Step 4: Create Flex Attributes," on page 317.

## Categorizing Image Assets for Display in Image Picker

Using the `CATEGORYATTRIBUTENAME` and `RESTRICTEDCATEGORYLIST` parameters described in the previous section, you can restrict an instance of Image Picker to display only selected categories of assets belonging to the selected image asset type. Before you do so, the following conditions must be satisfied:

1. You must add a category attribute of type string to the selected image asset type or flex definition that will store the category descriptor for each asset within the selected asset type. Image Picker will use the value of this attribute to generate a list of asset categories which it is allowed display. For instructions on creating attributes, see "Step 4: Create Flex Attributes," on page 317.
2. You or your content providers must fill in the category field for each asset of the selected image asset type, as appropriate. If an asset is not assigned a category, it will not be displayed by a category-restricted instance of Image Picker.

## Sample Image Picker Attribute Editor Definition Code

A sample XML definition for the Image Picker attribute editor is included below for your reference. Use it to get an idea of how to configure Image Picker on your system.

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT>
<PRESENTATIONOBJECT NAME="ImagePicker">
<IMAGEPICKER>
ASSETTYPENAME="Media_C"
ATTRIBUTETYPENAME="Media_A"
ATTRIBUTENAME="FSII_ImageFile"
CATEGORYATTRIBUTENAME="FSII_ImageCategory"
RESTRICTEDCATEGORYLIST="Audio,Video">
</IMAGEPICKER>
</PRESENTATIONOBJECT>
```

# Configuring eWebEditPro

eWebEditPro is a third-party HTML editor from Ektron, Inc. It is not delivered with Content Server. Installation and configuration instructions are included in the eWebEditPro release notes. For instructions on selecting an input method (such as eWebEditPro) for a field in an asset form, see “Step 4: Create Flex Attributes,” on page 317.

# Configuring the RENDERFLASH Attribute Editor

This section lists the parameters whose values you must specify in the XML definition when configuring an instance of the `RENDERFLASH` attribute editor.

For the explanation of the Flash content management model to which these parameters apply, see Chapter 30, "Setting Up Flash Content Management." The highlighted terms correspond to the names of data model components shown in Figure 14, on page 666.

| Parameter | Value |
|---|---|
| `FFTYPE` | Asset type of the **SWF asset**.<br>Example: `SWFAsset_C` |
| `FFATTRTYPE` | Asset type of the **SWF source file attribute** in the SWF asset.<br>Example: `Flash_A` |
| `FFATTRNAME` | Name of the **SWF source file attribute** in the SWF asset.<br>Example: `SWFSourceFile` |
| `IFTYPE` | Asset type of the **image asset**.<br>Example: `Media_C` |
| `IFATTRTYPE` | Asset type of the **image file attribute** in the image asset.<br>Example: `Media_A` |
| `IFATTRNAME` | Name of the **image file attribute** in the image asset.<br>Example: `FSII_ImageFile` |
| `FATYPE` | Asset type of the **Flash content asset**.<br>Example: `Flash_C` |
| `FAATTRTYPE` | Asset type of the **Flash content asset attributes** (that is, the **SWF asset**, **image asset**, and **text** attributes).<br>Example: `Flash_A` |
| `FAFLASHATTRNAME` | Name of the **SWF asset attribute** in the Flash content asset.<br>Example: `SWFAsset` |
| `FAIMAGEATTRNAME` | Name of the **image asset attribute** in the Flash content asset.<br>Example: `FlashImages` |
| `FATEXTATTRNAME` | Name of the **text attribute** in the Flash content asset.<br>Example: `FlashText` |

## Sample RENDERFLASH Attribute Editor Definition Code

An example RENDERFLASH attribute editor definition, based on the examples from previous sections, is shown below for your reference. Examine it to get an idea of how to configure the RENDERFLASH attribute editor on your system.

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT>
<PRESENTATIONOBJECT NAME="FlashPreview">
<RENDERFLASH
      FFATTRTYPE="Flash_A"
      FFATTRNAME="SWFSourceFile"
      FFTYPE="SWFAsset_C"
      IFATTRTYPE="Media_A"
      IFATTRNAME="FSII_ImageFile"
      IFTYPE="Media_C"
      FAATTRTYPE="Flash_A"
      FATYPE="Flash_C"
      FAFLASHATTRNAME="SWFAsset"
      FAIMAGEATTRNAME="FlashImages"
      FATEXTATTRNAME="FlashText">
</RENDERFLASH>
</PRESENTATIONOBJECT>
```

Chapter 20

# Importing Assets of Any Type

After you have determined your data design, created your asset types, tested them on your development system, and moved them to your management system, the next step is to import assets (content) from their current source in to the database on the management system. For example, you could be using a wire feed service or some other source of remotely generated content, and need to import that content into the Content Server database on your management system.

To import any data into the Content Server database, you can use the XMLPost utility. This utility is based on the Content Server FormPoster Java class and it is delivered with the Content Server base product. It imports data using the HTTP POST protocol.

This chapter describes the general process of importing assets with the XMLPost utility. You use the information in this chapter for importing assets of all types. The next chapter, Chapter 21, “Importing Flex Assets,”provides additional information that you need to import your assets when you are using the flex asset data model.

This chapter contains the following sections:


- The XMLPost Utility
- XMLPost Configuration Files
- XMLPost Source Files
- Using the XMLPost Utility
- Customizing RemoteContentPost and PreUpdate
- Troubleshooting XMLPost

# The XMLPost Utility

To import assets, you instruct the XMLPost utility to invoke one of the importing (posting) elements provided by CS-Direct or CS-Direct Advantage, as appropriate for that asset type.

There are four components involved in this process:

- The **XMLPost** utility, which is delivered with Content Server.
- A **posting element**. CS-Direct delivers a posting element named `RemoteContentPost`. CS-Direct Advantage delivers three additional posting elements, described in Chapter 21, "Importing Flex Assets."
- A **configuration file** with an `.ini` file extension. You create a configuration file for each asset type that you plan to import. This file contains information about what to expect in the source files (what tags XMLPost will find there), what to do with the data provided, and which importing (posting) element to use to import the data.
- **Source files**. You provide an individual source file for each asset that you want to import (well-formed XML files). Each tag in a file identifies a field for that asset type. The information contained in the tag is the data to be written to that column.

The XMLPost utility parses the configuration file to determine how to interpret the data provided for the asset type, parses the source files and creates name/value pairs for each field value, and passes those name/value pairs as ICS variables to the `RemoteContentPost` element. The `RemoteContentPost` element then creates the asset from the variables.

You can also create your own posting elements that work with the XMLPost utility. However, for importing assets, the posting elements that are provided by CS-Direct and CS-Direct Advantage should meet your needs.

**Note**

For added security, you can rename the `RemoteContentPost` page to prevent attempts to hack into the system.

## Overview

This section provides a brief overview of the steps that the developer completes before invoking the XMLPost utility and what the XMLPost utility does.

### What the Developer Does

When you import assets into your Content Server database, you perform four general steps:

1. You create a configuration file that identifies the type of asset that is to be imported and the tags that are used in the source files.

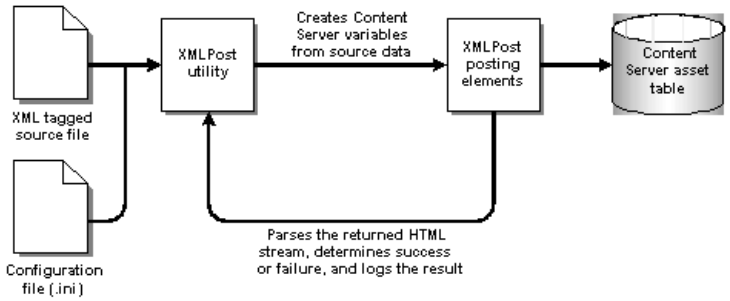
   This file also sets several configuration properties, including the name of the SiteCatalog entry for the posting element that you want XMLPost to use. For all assets, the name of this posting element is RemoteContentPost. For information about the posting elements for flex assets, see Chapter 21, "Importing Flex Assets."

Note that the configuration file is specific for this asset type. You must provide a separate configuration file for each asset type.

2. You create the source files for the data that you want to import. Note that you create a separate source file for each individual asset.
3. You place the source and configuration files in a directory on the management system.
4. From that directory, you invoke the XMLPost utility, identifying the source files and the configuration file to use for those source files.

## What XMLPost and Content Server Do

After you invoke the XMLPost utility to import the source files, this is what happens next, as shown in the following diagram and list of steps:




1. The XMLPost utility parses the configuration file.
2. XMLPost parses the source file and creates name/value pairs for each field value specified in the source file.
3. XMLPost invokes the FormPoster Java class by posting (HTTP POST) the name/value pairs as ICS variables to the page name passed in from the configuration file. When you are importing basic asset types, that pagename is:

   `OpenMarket/Xcelerate/Actions/RemoteContentPost`
4. Content Server locates the page in the `SiteCatalog` table and invokes the root element of the `RemoteContentPost` page, which has the same name by default (`RemoteContentPost`).
5. The `RemoteContentPost` element passes the data from the source files as variables to the `PreUpdate` element for assets of that type.
6. The `PreUpdate` element for assets of that type sets the variable values for that asset and then returns to the `RemoteContentPost` element.
7. The `RemoteContentPost` element creates the asset.
8. The web server returns a stream of HTML to XMLPost, which then parses the stream to determine whether the import operation succeeded or failed, logging the results to a text file that you specify in the configuration file.
9. If the asset type of the asset that you are importing uses a search engine, `RemoteContentPost` indexes the new element.
10. If you set a certain parameter in the configuration file, `RemoteContentPost` deletes the source files for the assets that were successfully imported.

# XMLPost Configuration Files

There are three types of properties in a configuration file for XMLPost:

- Properties that provide information to XMLPost about the database and environment. These properties remain the same even if you create your own posting element.
- Properties that provide configuration values for the posting (importing) process. This chapter describes the properties that you must provide for the `RemoteContentPost` element to function correctly.

  Examples of properties include the URL of the page that invokes `RemoteContentPost`, a user name and password that gives XMLPost write privileges to the asset type table in the database, the name of the asset type that you want to import, how to log errors, and any data values that are the same for all of the assets that you are importing.
- Properties that specify the tags that are used in the source files.

Certain information, such as which site the assets should belong to or which workflow should be assigned to the asset, can be configured either in the `RemoteContentPost` section of the configuration file or the source file section.

For example, if you have only one content management site or if all of the assets that you are importing belong to the same site, specify the name of the site in the configuration section so you do not have to repeat that information in each source file. If your system has more than one content management site, specify which sites an asset belongs to in the individual source files.

## Configuration Properties for XMLPost

The following table lists the properties that specify database connection information and other general configuration instructions that the XMLPost utility needs:

| Property | Description |
|---|---|
| `xmlpost.xmlfilenamefilter` | Required.<br>The file extension for your source files. Typically set to `xml`.<br>For example:<br>`xmlpost.xmlfilenamefilter: .xml` |
| `xmlpost.proxyhost` | Optional.<br>If a firewall separates you and the Content Server database that you want to import the assets in to, use this property to specify the host name of the proxy server.<br>For example:<br>`xmlpost.proxyhost: nameOfServer` |

| Property | Description |
|---|---|
| `xmlpost.proxyport` | Optional.<br>If a firewall separates you and the Content Server database that you want to import the assets in to, use this property to specify the port number on the proxy server that XMLPost should connect to.<br>For example:<br>`xmlpost.proxyport: 80` |
| `xmlpost.url` | Required.<br>The first part of the URL for the page entry of the posting element.<br>XMLPost creates the URL for the posting element by prepending the value specified for this property to the value specified for the pagename `postargname` (described below).<br>The value that you set for this property should use the following convention:<br>• The name of the server that holds the Content Server database.<br>• The CGI path appropriate for the application server software installed on the server. For example, for WebLogic, WebSphere, and Sun ONE this path is `/servlet/` and for iPlanet, this path is `/NASapp/cs/`.<br>• The name of the ContentServer servlet.<br>For example:<br>`xmlpost.url: http://`*`servername`*`/servlet/ContentServer`<br>or<br>`xmlpost.url: http://`*`servername`*`/NASapp/cs/ContentServer` |
| `xmlpost.logfile` | Optional.<br>The name of the file to log the results of importing (posting) each source file.<br>Each source file is posted to the Content Server database through a post request. When the post request returns from the web server, XMLPost parses the HTML stream that the web server returned, searching for the `postsuccess` and `postfailure` parameters. XMLPost then writes the result to the file that you name identify with this parameter.<br>For example:<br>`xmlpost.logfile: ArticlePost.txt` |

| Property | Description |
|---|---|
| `xmlpost.success` | Optional.<br>The string to look for in the response to determine if the post was a success.<br>For example:<br>`xmlpost.success: Success!` |
| `xmlpost.failure` | Optional.<br>The string to look for in the response to determine if the post was a failure.<br>For example:<br>`xmlpost.failure: Error` |
| `xmlpost.deletefile` | Optional.<br>Whether to delete the source files after they have been successfully imported into the Content Server database. Valid settings are `y` (yes) or `n` (no). By default, the source files are not deleted.<br>For example:<br>`xmlpost.deletefile: y` |

## Configuration Properties for the Posting Element

The following table lists the arguments that specify information that must be posted to the `RemoteContentPost` page (and passed to the `RemoteContentPost` element). The values of these arguments are concatenated into the URL that is posted to the `RemoteContentPost` page; they can be in any order in the configuration file:

| Property | Description |
|---|---|
| `xmlpost.numargs` | Required.<br>There are several required variables that the configuration file passes to XMLPost as name/value pairs attached to the URL, the primary of which is the page name. Use this property (`xmlpost.numargs`) to tell XMLPost how many variables the configuration file is passing in.<br>For example:<br>`xmlpost.numargs: 7`<br>Note that you can also specify your own custom variables with these name/value pairs. |
| `xmlpost.argname1: pagename` | Required.<br>The pagename for the `RemoteContentPost` element. Typically the `pagename` argument is specified as `xmlpost.argname1`.<br>For example:<br>`xmlpost.argname1: pagename`<br>`xmlpost.argvalue1: OpenMarket/Xcelerate/Actions/RemoteContentPost` |
| `xmlpost.argname2: AssetType` | Required.<br>The asset type of the assets that are defined in the source files. Typically, `AssetType` is specified as `xmlpost.argname2`.<br>For example:<br>`xmlpost.argname2: AssetType`<br>`xmlpost.argvalue2: Collection`<br>Note that the value for the `AssetType` argument must exactly match the table name of the table that holds assets of this type. |

| Property | Description |
|---|---|
| `xmlpost.argname3: authusername` | Required.<br>The user name that you want XMLPost to use to log into the Content Server database that you are importing the assets into. Typically, `authusername` is specified as `xmlpost.argname3`.<br>For example:<br>`xmlpost.argname3: authusername`<br>`xmlpost.argvalue3: editor`<br>The user name that you specify must have permission to write to the table that holds assets of the type that you are importing. (That is, it must have the appropriate ACLs assigned to it.) |
| `xmlpost.argname4: authpassword` | Required.<br>The password for the user that XMLPost logs in as to the Content Server database that you are importing the assets into. Typically, `authpassword` is specified as `xmlpost.argname4`.<br>For example:<br>`xmlpost.argname4: authpassword`<br>`xmlpost.argvalue4: xceleditor` |
| `xmlpost.argname5: xmlpostdebug` | Optional<br>Whether or not to include debugging information with the results information that is written to the XMLPost log file identified with the `xmlpost.logfile` property.<br>You can set this property to any value. For example:<br>`xmlpost.argname5: xmlpostdebug`<br>`xmlpost.argvalue5: on`<br>**Note:** Be sure to include a value for the `xmlpost.logfile` property if you enable debugging. |
| `xmlpost.argname6: inifile` | Optional.<br>The name of the `ini` file to use when connecting to the Content Server database. Typically, `inifile` is specified as `xmlpost.argname5`.<br>For example:<br>`xmlpost.argname6: inifile`<br>`xmlpost.argvalue6: futuretense.ini` |

| Property | Description |
|---|---|
| `xmlpost.argname7: publication` | Optional.<br>Athough using this property is optional, you must specify a site for each asset that you are importing.<br>If your system uses one content management site (publication) or if all assets of this type should be enabled on the same site, use this argument to set the name of the site.<br>For example:<br>`xmlpost.argname7: publication`<br>`xmlpost.argvalue7: Burlington Financial`<br>If your system uses more than one content management site, you must specify the value for site for each asset in the individual source files. See "Configuration Properties for the Source Files" on page 415 for more information. |
| `xmlpost.argname8: startmenu` | Optional.<br>If you are using workflow and you want the same workflow assigned to all of the assets that you are importing, use this argument to set the Start Menu shortcut for the assets. (It is a Start Menu shortcut that assigns a workflow ID to a new asset.)<br>For example:<br>`xmlpost.argname8: startmenu`<br>`xmlpost.argvalue8: New Article`<br>If you have more than one workflow for assets of this type, you must specify the value for the Start Menu shortcut for each asset in the individual source files. See "Configuration Properties for the Source Files" on page 415 for more information. |

## Configuration Properties for the Source Files

The source file section in a configuration file specifies which tags are used in the source files.

A tag represents a column name in the table that holds assets of this type. The content between a pair of tags is the information that is to be written to that column. Configuration files must list a tag for each column in the asset type's primary storage table, which is why you must provide a separate configuration file for each asset type.

### Site Properties

In addition to the tags that pertain to your asset types, there are four tags that you can use with all asset types to specify certain site configuration properties, that is, which sites an asset should be associated with and which workflow it should use. This table lists the site tags:

| Site tag property | Value | Description |
|---|---|---|
| `postpublication` | `y` or `n`<br>(yes or no) | Optional.<br>Specifies that a source file will provide a site name that identifies which site the asset belongs to.<br>For example:<br>postpublication: y<br>Note that a site (publication) value provided in a source file with the `publication` tag overrides the value specified for a publication argument in the XMLPost section of the configuration file. |
| `postprimarypubid` | `y` or `n`<br>(yes or no) | Optional.<br>Specifies that a source file will provide a value for pubid (a unique ID for the site) that identifies which site the asset belongs to.<br>For example:<br>postprimarypubid: y |
| `postpublist` | `y` or `n`<br>(yes or no) | Optional.<br>Specifies that a source file will provide a list of sites that the asset is shared with.<br>For example:<br>postpublist: y |
| `poststartmenu` | `y` or `n`<br>(yes or no) | Optional.<br>Specifies that a source file will provide a value for the Start Menu short cut that places the asset into a workflow process.<br>For example:<br>poststartmenu: y |

Remember that if the site or the workflow is the same for all of the assets that you are importing, you can specify the value for site or workflow as an argument in the XMLPost section of the configuration file. That way, you do not have to duplicate the same information in all of the source files.

## Asset Type Properties

To set up the tags that are specific to your asset types, you specify a tag for each column in the database table for assets of that type. However, the source files are not required to include data tagged with every tag in the configuration file. (Of course, they must include data for required fields.)

For each tag representing a field (column), you specify the name of the tag and optionally some additional processing properties for the tag. The name of the tag is the name of the

field (column). For the additional properties, the convention is a word prepended to the name of the tag.

The following table describes how to specify the tags that are specific to your asset types:

| Tag property | Value | Description |
|---|---|---|
| `post`*`tagname`* | `y` or `n`<br>(yes or no) | Required.<br>Specifies the name of the tag. The name should exactly match the name of the field that it represents.<br>For example, the tag property for a **name** field is:<br>`postname: y` |
| `trunc`*`tagname`* | *`N`*<br>(integer) | Optional.<br>Whether to truncate the data in the source file marked by this tag.<br>For example:<br>`truncname: 64`<br>By setting this property for the tag, if XMLPost finds a string in the `<name>` tag that exceeds 64 characters, it shortens it to 64 characters and stores the truncated string in the variable. |
| `notrim`*`tagname`* | `y` or `n`<br>(yes or no) | Optional.<br>Whether to trim the white space at the beginning or end of the marked with the tag. If you do not want the white space trimmed, set this property to `y` (yes).<br>For example:<br>`notrimname: y`<br>If you do not specify this property, XMLPost trims the white space for the tag by default. |

| Tag property | Value | Description |
|---|---|---|
| `multi`*`tagname`* | `combine`<br>or<br>*`separate`* | Required if the same tag is used more than once in a single source file.<br>Determines how many variables to use for the data when a tag is used more than once in the source file. If you set it to `combine`, the data from all of the tags is stored in the same variable with commas separating each value (a comma delimited string).<br>If you set it to `separate`, the data from each tag is stored in a separate variable. Those variables are identified by appending the value that you set for `seed`*`tagname`* to the variable name.<br>For example, for a `keyword` field (column):<br>• If you set `multikeyword: combine`, XMLPost stores all the values marked by a `keyword` tag to the same `keyword` variable.<br>• If you set `multikeyword: separate` and `seedkeyword: 1,` XMLPost stores each value in a separate variable. The first value it finds is stored in a variable named `keyword1`. The second value is stored in a variable named `keyword2`, and so on. |
| `seed`*`tagname`* | *`seed value`* | Required when `multi`*`tagname`* is set to `separate`.<br>The number to start at when XMLPost increments the suffix assigned to variable names when a tag is used more than once and you do not want the data contained in those tags written to the same variable. See the description of `multi`*`tagname`*.<br>For example:<br>`multikeyword: separate`<br>`seedkeyword: 1` |

| Tag property | Value | Description |
|---|---|---|
| `file`*`tagname`* | `y` or `n`<br>(yes or no) | Required if the tag represents an upload field (a URL column or BLOB).<br>If the tag represents a field that has a URL column, you must include this property and the source file must specify the name of the file that `RemoteContentPost` is to upload to that column.<br>For example, the imagefile asset type from the Burlington Financial sample site has an upload field named `urlpicture`. A configuration file for the imagefile asset type would have the following propeties:<br>`posturlpicture: y`<br>`fileurlpicture: y`<br>Then, in the source file for an imagefile asset, you specify the value for the `urlpicture` field like this:<br>`<urlpicture>`*`relative_path_to/filename.jpg`*<br>`</urlpicture>`<br>Note that you must specify the location of the file with a relative path—relative to the directory in which you are running the XMLPost utility. |

## Sample XMLPost Configuration File

Here is a sample configuration file named `imagefile.ini`, used to import imagefile assets for the Burlington Financial sample site.

```
xmlpost.xmlfilenamefilter: xml
#xmlpost.xmlproxypost: Future
#xmlpost.xmlproxyport: 80
xmlpost.url: http://localhost/servlet/ContentServer
xmlpost.numargs: 6
xmlpost.argname1: pagename
xmlpost.argvalue1: OpenMarket/Xcelerate/Actions/RemoteContentPost
xmlpost.argname2: AssetType
xmlpost.argvalue2: ImageFile
xmlpost.argname3: authusername
xmlpost.argvalue3: user_author
xmlpost.argname4: authpassword
xmlpost.argvalue4: user
xmlpost.argname5: inifile
xmlpost.argvalue5: futuretense.ini
xmlpost.argname6: publication
xmlpost.argvalue6: BurlingtonFinancial

xmlpost.success: Success
xmlpost.failure: Error
```

```
xmlpost.logfile: ImageFilePost.txt

postpublication: y
postprimarypubid: y
postpublist: y

postcategory: y
truncategory: 4

postpath: y
truncpath: 255

postname: y
truncname: 32

posttemplate: y
trunctemplate: 32

postsubtype: y
truncsubtype: 24

postfilename: y
truncfilename: 64

poststartdate: y

postdescription: y
truncdescription: 128

postsource: y

posturlpicture: y
fileurlpicture: y

posturlthumbnail: y
fileurlthumbnail: y

postmimetype: y
postwidth: y
postheight: y
postalign: y
postalttext: y

postkeywords: y
multikeywords: combine
trunckeywords: 128

postimagedate: y
```

# XMLPost Source Files

Source files must be made up of well-formed XML without the need for a document type definition (DTD) file. Actually, the configuration file functions something like a DTD file in that it defines the tags that will be processed in the source files.

The data in your source files must be tagged with tags whose names match the column names for the table that holds assets of that type. For example, a source file for a Burlington Financial imagefile asset uses tags named `name`, `caption`, `picutureurl`, and so on.

This chapter does not describe how to automate the generation of your XML source files. How you create your source files depends on the source of your data and the tools that you have to convert your data into XML files. This chapter describes what needs to be in your source files and what XMLPost does with them.

## Sample XMLPost Source File

Here is a sample source file for a Burlington Financial imagefile asset. Its tags are defined in the sample configuration file in the previous section, "Sample XMLPost Configuration File" on page 419:

```
<document>
<name>High Five 25</name>
<keyword>Five</keyword>
<category>a</category>
<artist>by Ann. Artist</artist>
<alttext>Congratulations</alttext>
<align>CENTER</align>
<caption>A man extends <keyword>congratulations</keyword> with a
boy.</caption>
<pictureurl>/images/eZine/highfive.jpg</pictureurl>
</document>
```

### How the Data is Passed (Posted)

All of the text contained between a pair of XML tags in a source file is passed to the `RemoteContentPost` element from XMLPost as a variable that uses the `Variables.`*`tagname`* syntax convention.

For example, this line of code:

```
<name>High Five 25</name>
```

is sent to `RemoteContentPost` as `Variables.name` and the value of `name` is the string "High Five".

## XMLPost and File Encoding

If the data in a source file does not use the Content Server system's default file encoding but the database can accommodate that character set, you can specify the alternate file encoding in the XML version statement at the beginning of the file.

For example:

```
<?xml version= "1.0" encoding="UTF-8" ?>
```

# Using the XMLPost Utility

You can invoke the XMLPost utility in one of many ways:

- From the command line
- From a script or batch file
- From a program

No matter how you start XMLPost, you must provide the following pieces of information:

- The name of the configuration file to use
- The source files, which can be specified as a single file, a list of files, or a directory of files

## Before You Begin

- Before you can use the XMLPost utility, the following must be true:
  - Your asset types are created. (Otherwise, there are no database tables to import the assets into.)
  - Your content management sites are created and the appropriate asset types are enabled for each site.
  - If you are using workflow, your workflow processes are created.
  - Your Start Menu shortcuts are created and, if you are using workflow, they assign the appropriate workflow process to the appropriate asset types.
  - The templates for the asset type are created.
  - The association fields for the asset types are created. However, to use XMLPost to set the value of an association field requires custom code. See "Customizing RemoteContentPost and PreUpdate" on page 426 for more information.
- When invoking XMLPost, include the following command before the classpath to ensure UTF-8 encoding: **`-Dfile.encoding=UTF-8`**

> **Note**
>
> Check all `.jar` files for version number, which can differ from one Content Server patch to the next.

## Running XMLPost from the Command Line

Complete the following steps:

1. Place the configuration file and source files in a directory on a system that has Content Server installed.
2. Run the following command (on a single command line) from that directory:

Windows:

```
java -Xmx512m -Dfile.encoding=UTF-8 -classpath
   <cs_app_dir>\WEB-INF\lib\commons-logging-1.1.1.jar;
   <cs_app_dir>\WEB-INF\lib\commons-httpclient-3.1.jar;
   <cs_app_dir>\WEB-INF\lib\commons-lang-2.4.jar;
   <cs_app_dir>\WEB-INF\lib\commons-codec-1.4.jar;
   <cs_app_dir>\WEB-INF\lib\commons-collections-3.1.jar;
   <cs_app_dir>\WEB-INF\lib\commons-io-1.3.1.jar;
   <cs_app_dir>\WEB-INF\lib\MSXML.jar;
   <cs_app_dir>\WEB-INF\lib\cs.jar;
   <cs_app_dir>\WEB-INF\lib\cs-core.jar;
   <j2eeSDK_dir>/j2ee.jar COM.FutureTense.XML.Post.XMLPostMain
      -sSourcefile.xml -cConfigfile.ini
```

Solaris:

```
java -Xmx512m -Dfile.encoding=UTF-8 -classpath
   <cs_app_dir>\WEB-INF\lib\commons-logging-1.1.1.jar:
   <cs_app_dir>\WEB-INF\lib\commons-httpclient-3.1.jar:
   <cs_app_dir>\WEB-INF\lib\commons-lang-2.4.jar:
   <cs_app_dir>\WEB-INF\lib\commons-codec-1.4.jar:
   <cs_app_dir>\WEB-INF\lib\commons-collections-3.1.jar:
   <cs_app_dir>\WEB-INF\lib\commons-io-1.3.1.jar:
   <cs_app_dir>\WEB-INF\lib\MSXML.jar:
   <cs_app_dir>\WEB-INF\lib\cs.jar:
   <cs_app_dir>\WEB-INF\lib\cs-core.jar:
   <j2eeSDK_dir>/j2ee.jar COM.FutureTense.XML.Post.XMLPostMain
      -sSourcefile.xml -cConfigfile.ini
```

where:

- `-Xmx512m` sets the maximum memory to use, which is needed with larger inserts
- `<cs_install_dir>` is the directory where Content Server is installed.
- `<cs_app_dir>` is the directory on your application server where the CS application has been deployed.
- `<j2eeSDK_dir>` is the directory where your J2EE SDK is installed.

(Note that there are several options for designating the source file. See "Options for Identifying Source Files" on page 424 for information.)

**Note**

- The `j2ee.jar` file is part of the J2EE SDK. You must install the SDK before running XMLPost.
- If the source files and configuration file are not in the directory that you are working in, you must provide the path to those files in the command line. For example: `-s/products/product.xml`.

## Options for Identifying Source Files

The source parameter that you use to identify the source files to the XMLPost utility can point to any of the following:

- A single file.
- A directory of files. All the files in that directory that have the file extension (typically `.xml`) designated by the configuration file will be posted (imported).
- A list file that provides a list of all the files that you want to import. It is similar to an `.ini` file but it has a file extension of `.lst`.

### A Single File

To post the contents of one file, specify the name of that file in the command line. The following example (Solaris) instructs XMLPost to use a configuration file named `articlepost.ini` and one source file named `article.xml`:

```
java -Xmx512m -Dfile.encoding=UTF-8 -classpath
   <cs_app_dir>\WEB-INF\lib\commons-logging-1.1.1.jar:
   <cs_app_dir>\WEB-INF\lib\commons-httpclient-3.1.jar:
   <cs_app_dir>\WEB-INF\lib\commons-lang-2.4.jar:
   <cs_app_dir>\WEB-INF\lib\commons-codec-1.4.jar:
   <cs_app_dir>\WEB-INF\lib\commons-collections-3.1.jar:
   <cs_app_dir>\WEB-INF\lib\commons-io-1.3.1.jar:
   <cs_app_dir>\WEB-INF\lib\MSXML.jar:
   <cs_app_dir>\WEB-INF\lib\cs.jar:
   <cs_app_dir>\WEB-INF\lib\cs-core.jar:
   <j2eeSDK_dir>/j2ee.jar COM.FutureTense.XML.Post.XMLPostMain
      -sarticle.xml -carticlepost.ini
```

### A Directory of Files

To post all the files in a directory, specify the path to that directory in the command line. The following example (Solaris) instructs XMLPost to import the files in the `xmlpostfiles` directory:

```
java -Xmx512m -Dfile.encoding=UTF-8 -classpath
   <cs_app_dir>\WEB-INF\lib\commons-logging-1.1.1.jar:
   <cs_app_dir>\WEB-INF\lib\commons-httpclient-3.1.jar:
   <cs_app_dir>\WEB-INF\lib\commons-lang-2.4.jar:
   <cs_app_dir>\WEB-INF\lib\commons-codec-1.4.jar:
   <cs_app_dir>\WEB-INF\lib\commons-collections-3.1.jar:
   <cs_app_dir>\WEB-INF\lib\commons-io-1.3.1.jar:
   <cs_app_dir>\WEB-INF\lib\MSXML.jar:
   <cs_app_dir>\WEB-INF\lib\cs.jar:
   <cs_app_dir>\WEB-INF\lib\cs-core.jar:
   <j2eeSDK_dir>/j2ee.jar COM.FutureTense.XML.Post.XMLPostMain
      -sxmlpostfiles -carticlepost.ini
```

### A List File

As an alternative to specifying a directory, you can create a list file that uses the format of an `.ini` file and includes the following properties:

- `numfiles`, which specifies how many files are included in the list.
- `file`*N*, which specifies the path to a file and its file name. The *N* stands for the file's order in the list file. The first file listed is `file1`, the second is `file2`, and so on.

The value of *N* for the last `file`*N* in the list must match the value specified by the `numfiles` property. XMLPost stops importing when it has imported as many files as it is told to expect by the `numfiles` property. If you have included more files than `numfiles` states, XMLPost does not import them.

The file extension for a list file must be `.lst`.

Here is an example list file, named `xmlpostfiles.lst`:

```
numfiles: 3
file1: c:\xmlpost\article1.xml
file2: c:\xmlpost\article2.xml
file3: c:\xmlpost\article3.xml
```

To post the files referenced in this file list, specify the name of the list file in the command line. The following example (Windows NT) instructs XMLPost to import the files specified in the `xmlpostfiles.lst` file:

```
java -Xmx512m -Dfile.encoding=UTF-8 -classpath
   <cs_app_dir>\WEB-INF\lib\commons-logging-1.1.1.jar;
   <cs_app_dir>\WEB-INF\lib\commons-httpclient-3.1.jar;
   <cs_app_dir>\WEB-INF\lib\commons-lang-2.4.jar;
   <cs_app_dir>\WEB-INF\lib\commons-codec-1.4.jar;
   <cs_app_dir>\WEB-INF\lib\commons-collections-3.1.jar;
   <cs_app_dir>\WEB-INF\lib\commons-io-1.3.1.jar;
   <cs_app_dir>\WEB-INF\lib\MSXML.jar;
   <cs_app_dir>\WEB-INF\lib\cs.jar;
   <cs_app_dir>\WEB-INF\lib\cs-core.jar;
   <j2eeSDK_dir>/j2ee.jar COM.FutureTense.XML.Post.XMLPostMain
      -sc:\xmlpostfiles.lst -carticlepost.ini
```

## Running XMLPost as a Batch Process

When you want to import assets of more than one type, you must identify a unique configuration file for each asset type and therefore run XMLPost individually for each asset type. You can run XMLPost either manually, or automatically from a batch file. In the batch file, include a command line statement for each asset type (the statement identifies the configuration file and the location of the source files). You can use any of the ways described in the preceding section to identify the source files.

## Running XMLPost Programmatically

You can also invoke the XMLPost utility programmatically by creating an XMLPost object and calling the `doIt` method `doIt(String[] args)`, where the input is a string array. The elements of the array are the same flags that you use when running XMLPost from the command line.

For example:

```
String args [] = {"-sSourcefile.xml","-cConfigfile.ini"};
COM.FutureTense.XML.Post.XMLPost poster = new
COM.FutureTense.XML.Post.XMLPost();
try {
poster.doIt(args);
} catch (Exception e) {
```

```
ics.LogMsg("error in XMLPost under program control");
}
```

Note that you must include the complete path to the source files and the configuration file.

# Customizing RemoteContentPost and PreUpdate

If necessary, you can customize the XMLPost process by adding or modifying code in the `RemoteContentPost` element or the `PreUpdate` element for your asset types.

If you want to import information about an asset to other tables, you must modify the `PreUpdate` element for that asset type.

This section provides two customization examples:

- Customizing the `PreUpdate` element for the article asset type so that it sets headline information in the description field. There is a description column in the `Article` table but the field in the "New" or "Edit" article form is called **Headline**.
- Customizing the `PreUpdate` element for the article asset type so that it can add associations to articles.

## Setting a Field Value Programmatically

The article asset type has a special condition: it has a field in the **New** and **Edit** forms called **Headline**, but the value for **Headline** is stored in the `description` column in the `Article` table. In order for headline text to be written to the correct column in the `Article` table when an article asset is imported (that is, the `description` column), the `PreUpdate` element for the article asset type was modified.

First, examine the sample configuration file named `ArticlePost.ini` that is located in the `Xcelerate/Samples/XMLPost` directory in your CS-Direct product kit. It has a tag specified for the **Headline** field:

```
# headline gets stored in the description field
postheadline: y
```

The following code in the `PreUpdate` element for the article asset type writes the data that `RemoteContentPost` passes in as `Variable.headline` to the correct database column:

```
<if COND="IsVariable.headline=true">
<then>
   <ASSET.SET NAME="theCurrentAsset" FIELD="description"
   VALUE="Variables.headline"/>
</then>
</if>
```

This example uses a tag called `ASSET.SET`. This tag sets data in a field for the asset that is currently in memory. It takes three parameters:

- `NAME` (required). The name of the asset object that is in memory. This asset object must have been previously instantiated either with the `ASSET.LOAD` tag or the `ASSET.CREATE` tag. By convention, CS-Direct uses the name `theCurrentAsset` to refer to the current asset object.

- `FIELD` (required). The name of the field whose value you want to set. The name of this field must exactly match the name of a column in the storage table for assets of this type.
- `VALUE` (required). The data to be inserted in the column.

## Setting an Asset Association

If an asset has an association with another asset, that information is written to the `AssetRelationTree` table. Because the standard behavior of XMLPost is to write asset information to the primary storage table of the asset type only, you must modify the `PreUpdate` element for the asset type if you want to specify asset associations.

For example, the article asset type has an association field named **MainImageFile.** When a content provider creates an article asset, she selects the appropriate imagefile asset in this field.

Examine the sample configuration file named `ArticlePost.ini` that is located in the `Xcelerate/Samples/XMLPost` directory in your CS-Direct product kit. It has a tag specified for the **MainImageFile** association field:

```
postMainImageFile-name: y
```

The following code in the `PreUpdate` element for the article asset type writes the data that `RemoteContentPost` passes in as `Variable.mainimagefile` to the correct database table:

```
<if COND="IsVariable.MainImageFile-name=true">
<then>
   <ASSET.LOAD NAME="anAssociatedImage" TYPE="ImageFile"
   FIELD="name" VALUE="Variables.MainImageFile-name"/>
   <if COND="IsError.Variables.errno=false">
         <then>
         <ASSET.GET NAME="anAssociatedImage" FIELD="id"
         OUTPUT="imageid"/>
         <ASSET.ADDCHILD NAME="theCurrentAsset" TYPE="ImageFile"
         CHILDID="Variables.imageid" CODE="MainImageFile"/>
         </then>
   </if>
</then>
</if>
```

**Note**

The `ASSET.ADDCHILD` tag creates only the link between the two assets; it does not create the associated asset. In order for this code to work, the asset specified with the `CHILDID` parameter must already exist in the Content Server database.

This example uses a tag named `ASSET.ADDCHILD`. This tag associates a child asset with the asset that is currently held in memory. It takes five parameters:

- `NAME` (required). The name of the asset object that is in memory. This asset object must have been previously instantiated either with the `ASSET.LOAD` tag or the `ASSET.CREATE` tag. By convention, CS-Direct uses the name `theCurrentAsset` to refer to the current asset object.

- `TYPE` (required). The asset type of the child asset.
- `CHILDID` (required). The ID of the child asset.
- `CODE` (optional). The name of the association. This value is written to the `ncode` column in the `AssetRelationTree` table.
- `RANK` (optional). A numeric value to establish an order for the child assets. This value is written to the `nrank` column in the `AssetRelationTree` table.

For information about `ASSET.GET` and `ASSET.LOAD`, see the *Content Server Tag Reference*.

# Troubleshooting XMLPost

This is a brief list of some possible problems that can occur when you run the XMLPost utility.

### XMLPost does not run and does not create a log file message

There are two possible reasons for XMLPost to not start:

- An invalid server name has been specified in the `xmlpost.URL` property setting in your configuration file.
- Content Server is not running on the system you are importing to. Start it.

### XMLPost fails and there is a “Missing Entity” statement in the log file

When you see this message in the log file, it means that there is invalid XML in the source file. Typically, your XML includes HTML code and that code includes special HTML characters that are not referred to by their character entity codes. For best coding practice, embed any HTML code in a `<![CDATA[...]]>` tag.

### Error 105 is triggered when XMLPost tries to save an asset

There are several reasons why saving an asset can cause a database error.

One common reason for a 105 error is XMLPost trying to save data that is too large for the column (field). Resolving this depends on your goals. If it is acceptable for XMLPost to truncate the data that doesn’t fit into the column, you can add a `trunc`*`tag`* property to the configuration file. For example, `truncbody: 2000`.

Another common reason for this error code is that an asset of that type with the same name already exists. Try changing the name of the asset and importing the asset again.

## Debugging the Posting Element

If you have modified the `RemoteContentPost` element in any way or have created your own posting element, you can use the XML Debugger utility to test it before you use it.

To use XML Debugger, replace `ContentServer` with `DebugServer` in the `xmlpost.url` property setting.

For example, change `xmlpost.url: http://6ipjk/servlet/ContentServer`

to `xmlpost.url: http://6ipjk/servlet/DebugServer`

For more information about the XML Debugger utility, see Chapter 8, “Content Server Tools and Utilities.”

Chapter 21

# Importing Flex Assets

Chapter 20, "Importing Assets of Any Type" presents the core information about using the XMLPost utility. If your Content Server configuration includes CS-Direct Advantage and you are using the flex asset data model, you have more tools for importing your assets.

CS-Direct Advantage provides three additional posting elements for the XMLPost utility and a bulk processing utility named BulkLoader.

This chapter describes the additional posting elements for the XML-Post utility. It contains the following sections:

- Overview
- XMLPost and the Flex Asset Model
- Importing the Structural Asset Types in the Flex Model
- Importing Flex Assets with XMLPost
- Editing Flex Assets with XMLPost
- Deleting Assets with XMLPost

This chapter refers to the BulkLoader utility, but for in-depth information about how to run it, see Chapter 22, "Importing Flex Assets with the BulkLoader Utility."

# Overview

CS-Direct Advantage provides two methods for importing assets that use the flex data model into the Content Server database:

- XMLPost. CS-Direct Advantage provides three additional posting elements that work with XMLPost: `addData`, `modifyData`, and `deleteData`.
- The BulkLoader utility.

## Importing the Data Structure Flex Asset Types

Before you can use either method, you must first create or import the data design or "structural" asset types into your flex families with XMLPost and the standard posting element, `RemoteContentPost`, provided by the CS-Direct product. That is, first you create or import the attribute editors, flex attributes, flex definitions, and flex parent definitions with the standard XMLPost posting element. If you are using the BulkLoader utility, the flex parents must also be imported with XMLPost or created.

## Importing the Flex Assets

After you import your data structure asset types, then you can import your flex assets with one of the two import methods, depending on the situation:

- Use BulkLoader to import a large number—thousands or hundreds of thousands—of flex assets.
- Use the CS-Direct Advantage posting element to load a moderate number—hundreds—of flex and flex parent assets.

### When to Use BulkLoader

When working within the basic asset model, it is typical to use XMLPost to import assets into the database on the management system and then publish those assets to the delivery system. This methodology changes with flex assets because the volume of data involved in a flex asset data model tends to be much greater than that in a basic asset model.

You use the BulkLoader utility during the initial setup of your Content Server system. See Chapter 22, "Importing Flex Assets with the BulkLoader Utility."

### When to Use XMLPost

For regular or incremental updates after the initial setup of your Content Server system—perhaps some or all of your data originates in an ERP system, for example—you use the XMLPost utility and the `addData` posting element.

## Importing Flex Assets: The Process

Because assets using the flex model have dependencies on each other, flex asset types must be imported in a specific sequence. And, as with basic assets, the asset types must exist, there must be sites created, and so on before you can use XMLPost to import assets.

For information about the basic prerequisites for using XMLPost that apply to all asset types (both asset models), see "Before You Begin" on page 422.

After those basic requirements are met, you must import your flex asset types into the Content Server database on the management system in the following sequence:

1. Attribute editors are optional, but if you plan to use them you must either import them or create them before you import your flex attributes. The configuration file must instruct XMLPost to call the `RemoteContentPost` element. For information, see "Attribute Editors" on page 434.
2. Flex attributes. The configuration file must instruct XMLPost to call the `RemoteContentPost` element. For information, see "Flex Attributes" on page 436.
3. Flex parent definitions. The configuration file must instruct XMLPost to call the `RemoteContentPost` element. For information, see "Flex Definitions and Flex Parent Definitions" on page 440.

   Flex definitions. The configuration file must instruct XMLPost to call the `RemoteContentPost` element. For information, see "Flex Definitions and Flex Parent Definitions" on page 440.

   **Note**

   You must import the flex parent definitions in the proper order. That is, if a parent definition refers to another parent definition asset, the referenced asset must already exist in the database.

   It is typical to import parent definitions one hierarchical level at a time, starting with the top level definitions.

4. Flex parent assets. Do one of the following:
   - If you are going to use XMLPost to import the flex assets, you can either import the flex parents individually or you can import them as part of the flex family tree for a flex assets.
   - If you are going to use the BulkLoader utility to import the flex assets, you must first use XMLPost to import the flex parent assets. The configuration file must instruct XMLPost to call the `RemoteContentPost` element. The file cannot specify the `addData` element because you are importing the parents without the entire family tree for the flex assets.

   For information, see "Flex Parents" on page 444.
5. (Optional) If you plan to use the BulkLoader utility to import flex assets into both the management system and the delivery system, you must first approve and publish all of the structural assets (attribute editors, flex attributes, flex definitions, parent definitions, and flex parents) from the management system to the delivery system.
6. Flex assets. Do one of the following:
   - Use the BulkLoader utility. See Chapter 22, "Importing Flex Assets with the BulkLoader Utility."
   - Use XMLPost. See "Importing Flex Assets with XMLPost" on page 445.

You must follow the sequence outlined in the preceding steps because there are dependencies built in to the data structure of a flex asset family. Additionally, note the following dependencies:

- If you have attributes of type `asset` and a flex parent or flex asset has such an attribute, the asset that you designate as the value of that attribute field must have already been created or imported.

- An asset that you set as the value for an attribute of type `asset` must be of the correct asset type.

## Using Custom Data Delimiters

If the data you are importing, editing, or deleting via XMLPost uses a different data delimiting schema than the Content Server default schema (see the table below for CS-default delimiter characters), you can specify custom delimiters as explained in the following table.

| Tag and property | Description |
|---|---|
| tag:<br>`<_xmlnamevaldelim_>`<br>property:<br>`post_xmlnamevaldelim_` | Optional.<br>Allows you to specify a custom character for delimiting name/value pairs from one another.<br>To specify a custom delimiter:<br>**1.** Set the property to `y` in the configuration file.<br>**2.** Use the tag in your XML file to define the custom delimiter. For example, to use the "at" character as a delimiter:<br>`<_xmlnamevaldelim_>@</_xmlnamevaldelim_>`<br>The default delimiter is the colon ( `:` ). |
| tag:<br>`<_xmlpostequaldelim_>`<br>property:<br>`post_xmlpostequaldelim_` | Optional.<br>Allows you to specify a custom character for delimiting attribute names from their values.<br>To specify a custom delimiter:<br>**1.** Set the property to `y` in the configuration file.<br>**2.** Use the tag in your XML file to define the custom delimiter. For example, to use two equal signs as a delimiter:<br>`<_xmlpostequaldelim_>==</_xmlpostequaldelim_>`<br>The default delimiter is the equal sign ( `=` ). |
| tag:<br>`<_xmlpostmulvaldelim_>`<br>property:<br>`post_xmlpostmulvaldelim_` | Optional.<br>Allows you to specify a custom character for delimiting the values of a multivalued attribute from one another.<br>To specify a custom delimiter:<br>**1.** Set the property to `y` in the configuration file.<br>**2.** Use the tag in your XML file to define the custom delimiter. For example, to use a hyphen as a delimiter:<br>`<_xmlpostmulvaldelim_>-</_xmlpostmulvaldelim_>`<br>The default delimiter is the semicolon ( `;` ). |

# XMLPost and the Flex Asset Model

The XMLPost utility works the same no matter which asset model or Content Server product you are using. However, CS-Direct Advantage provides additional processing logic in some of its standard elements for the flex asset types to support XMLPost because flex assets store their data in more than one database table (unlike basic asset types, which have one database table).

Additionally, CS-Direct Advantage provides both a posting element that enables you to use XMLPost to edit flex assets (`modifyData`) and a posting element that enables you to use XMLPost to delete assets of any type (`deleteData`).

This chapter provides additional information about creating configuration and source files specifically for the asset types in a flex family (and attribute editors). Be sure to also read Chapter 20, "Importing Assets of Any Type" for basic information that pertains to all XMLPost configuration and source files.

In the flex asset model, you specify a different posting element based on the following categories of asset types:

- Structural asset types that give the flex asset type and flex parent asset type their data structure. That is, attribute editors, attributes, flex definitions, and flex parent definitions.

  Use the standard CS-Direct posting element `RemoteContentPost` to import the structural asset types. (You cannot use the `addData` element with assets of these types.)

- Flex and flex parent asset type (for example, the product and product parent types in the GE Lighting sample site).

  Depending on the situation, you can use either the CS-Direct Advantage posting element `addData` to import the flex and flex parent asset types or the CS-Direct posting element `RemoteContentPost`. (See "Flex Parents" on page 444 and "Importing Flex Assets with XMLPost" on page 445 for information about which posting element to use.)

In both cases, you create configuration files and source files (as described in "Overview" on page 430 and supplemented in this chapter), and then invoke the XMLPost utility (as described in "Using the XMLPost Utility" on page 422).

**Note**

For reference, sample XMLPost code is provided **in** the Content Server installer package, in the **`/Xcelerate/Samples/XMLPost`** directory. The same folder contains the `readme.txt` file that describes the sample files.

## Internal Names vs. External Names

When you create your flex family of asset types (see "Step 1: Create a Flex Family" on page 313), you specify both an internal and an external name for your asset types.

The internal name is used for the primary storage table in the database. The external name is used in the CS-Direct Advantage "New," "Edit," and "Inspect" forms, in search results list, and so on. For example, the internal name for the attribute editor asset type is AttrTypes but that name is not used in the user interface. And the internal name for the GE Lighting

sample site's article asset type is `AArticles` but that name is not used in the user interface.

Because XMLPost communicates with the database, you must always use the internal name of the asset type in the configuration files and source files. For example, in a configuration file for attribute editors, you would specify the following:

```
postargname2: AssetType
postargvalue2: AttrTypes
```

# Importing the Structural Asset Types in the Flex Model

All of the information about configuration and source files for basic assets that is presented in Chapter 20, "Importing Assets of Any Type" applies to the configuration and source files for the flex asset types.

Additionally, this section provides example configuration and source files for the structural flex asset types.

## Attribute Editors

Attribute editors store their data in one table, named `AttrTypes`. AttrTypes is the internal name of the attribute editor asset type. Be sure to use this name in your configuration file for attribute editors.

The following table describes the configuration file properties and source file tags that you use with attribute editors:

| Attribute editor tag and property | Description |
|---|---|
| tag:<br>`<name>`<br>property:<br>`postname` | Required.<br>Name of the attribute editor asset; this is a required value for all asset types. Attribute names are limited to 64 characters and cannot contain spaces. |
| tag:<br>`<description>`<br>property:<br>`postdescription` | Optional.<br>Description of the use or function of the attribute. |
| tag: `<AttrTypeText>`<br>property:<br>`postAttrTypeText` | Required.<br>Either the name of the file with the attribute editor XML code, or the actual code.<br>This tag corresponds to the **XML in file** field and **Browse** button and the **XML field** in the **New** and **Edit** attribute editor forms in the Content Server interface. |

## Sample Configuration File: Attribute Editor

This is a sample configuration file for the attribute editor asset type. It works with the sample source file immediately following this example.

```
xmlpost.xmlfilenamefilter: .xml
xmlpost.url: http://izod19/servlet/ContentServer
xmlpost.numargs: 6
xmlpost.argname1: pagename
xmlpost.argvalue1: OpenMarket/Xcelerate/Actions/RemoteContentPost
xmlpost.argname2: AssetType
xmlpost.argvalue2: AttrTypes
# notice that you use the internal name of the asset type

xmlpost.argname3: authusername
xmlpost.argvalue3: user_editor
xmlpost.argname4: authpassword
xmlpost.argvalue4: user
xmlpost.argname5: inifile
xmlpost.argvalue5: futuretense.ini
xmlpost.argname6: startmenu
xmlpost.argvalue6: New Attribute Editor

xmlpost.success: Success
xmlpost.failure: Error
xmlpost.logfile: attreditorpostlog.txt

xmlpost.deletefile: y

postpublication: y

postname: y
postdescription: y
postAttrTypeText: y
```

## Sample Source File: Attribute Editor

The following source file is tagged for importing a check box attribute editor, or presentation object, for the GE Lighting sample catalog. It works with the preceding sample configuration file.

```
<document>
<publication>GE Lighting</publication>
<name>Editor4-CheckBoxes</name>
<description>Attribute Type Four Check Box</description>
<AttrTypeText>
<![CDATA[
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT SYSTEM "presentationobject.dtd">
   <PRESENTATIONOBJECT NAME="CheckBoxTest">
   <CHECKBOXES LAYOUT="VERTICAL">
      <ITEM>Red</ITEM>
      <ITEM>Green</ITEM>
      <ITEM>Blue</ITEM>
```

```
   </CHECKBOXES>
</PRESENTATIONOBJECT>
]]>
</AttrTypeText>
</document>
```

## Flex Attributes

Flex attributes have several tables but XMLPost writes to only two of them: the main storage table (for example, the `PAttributes` table for the GE Lighting sample site) and the attribute asset type's `_Extension` table (`PAttributes_Extension`, for example).

This means that the source file section of the configuration file must specify and the source file itself must use tags that represent columns in both tables. Those source file tags and configuration file properties are as follows:

| Flex attribute tag and property | Description |
|---|---|
| tag:<br>`<name>`<br>property:<br>`postname` | Required.<br>Name of the attribute; this is a required value for all asset types. Attribute names are limited to 64 characters and cannot contain spaces. |
| tag:<br>`<description>`<br>property:<br>`postdescription` | Optional.<br>Description of the use or function of the attribute. |
| tag:<br>`<valuestyle>`<br>property:<br>`postvaluestyle` | Optional.<br>Whether the attribute can hold a single value (`S`) or multiple values (`M`). If no this tag is not used, the attribute is set to hold a single value by default.<br>The attribute data type of `url` has been deprecated and replaced with the `blob` type in the 4.0 version of CS-Direct Advantage. If you are still using the `url` data type, note that you cannot specify `M` if the data type is `url`. |
| tag:<br>`<type>`<br>property:<br>`posttype` | Required.<br>The data type of the attribute. Valid options are `asset`, `date`, `float`, `int`, `money`, `string`, `text`, or `blob`. For definitions of these data types, see "Data Types for Attributes" on page 199. |
| tag: `<assettypename>`<br>property:<br>`postassettypename` | Required if `<type>` is set to `asset`.<br>The name of the asset type that the attribute holds. |

| Flex attribute tag and property | Description |
|---|---|
| tag:<br>`<upload>`<br>property:<br>`postupload` | Required if `<type>` is set to `blob` (or `url`, which is deprecated in 4.0).<br>The path to the directory that you want to store the attribute values in. Note that the value that you enter in this field is appended to the value set as the default storage directory (defdir) for the attribute table by the `cc.urlattrpath` property in the `gator.ini` file (which is `FutureTense/futuretense_cs/ccurl/` by default). |
| tag: `<attributetype>`<br>property:<br>`postattributetype` | Optional.<br>The name of the attribute editor to use, if applicable. |
| tag:<br>`<enginename>`<br>property:<br>`postenginename` | Optional.<br>If you are using a search engine on your management system, the name of the search engine. |
| tag:<br>`<charsetname>`<br>property:<br>`postcharsetname` | Optional.<br>The search engine character set to use. By default, it is set to ISO 8859-1. |
| tag:<br>`<editing>`<br>property:<br>`postediting` | Foreign attributes only.<br>Whether a foreign attribute can be edited through the CS-Direct Advantage forms (`L`), or edited externally using a third-party tool (`R`). `L` is the default. |
| tag:<br>`<storage>`<br>property:<br>`poststorage` | Foreign attributes only.<br>Whether the values for a foreign attribute are to be stored in a `_Mungo` table in the Content Server database (`L`) or in a foreign table (`R`). `L` is the default. |
| tag:<br>`<externalid>`<br>property:<br>`postexternalid` | Foreign attributes only.<br>The name of the column that serves as the primary key for the table that holds this foreign attribute; that is, the column that uniquely identifies the attribute. |
| tag:<br>`<externalcolumn>`<br>property:<br>`postexternalcolumn` | Foreign attributes only.<br>The name of the column in the foreign table that holds the values for this attribute. |

| Flex attribute tag and property | Description |
|---|---|
| tag:<br>`<externaltable>`<br>property:<br>`postexternaltable` | Foreign attributes only.<br>The name of the foreign table that contains the columns identified by externalid and external column. |
| tag:<br>`<publication>`<br>property:<br>`postpublication` | Optional.<br>The names of all the sites that can use this attribute. |

## Sample Configuration File: Flex Attribute

This is sample configuration file for the product attribute asset type from the GE Lighting sample site. It works with the sample source file immediately following this example.

```
xmlpost.xmlfilenamefilter: .xml

xmlpost.url: http://izod19/servlet/ContentServer
xmlpost.numargs: 6
xmlpost.argname1: pagename
xmlpost.argvalue1: OpenMarket/Xcelerate/Actions/RemoteContentPost
xmlpost.argname2: AssetType
xmlpost.argvalue2: PAttributes
# Notice that this is the internal name of the asset
# type. The external name of this asset type is
# Product Attribute.

xmlpost.argname3: authusername
xmlpost.argvalue3: user_editor
xmlpost.argname4: authpassword
xmlpost.argvalue4: user
xmlpost.argname5: inifile
xmlpost.argvalue5: futuretense.ini
xmlpost.argname6: startmenu
xmlpost.argvalue6: New Product Attribute

xmlpost.success: Success
xmlpost.failure: Error
xmlpost.logfile: attributespostlog.txt

xmlpost.deletefile: y

postpublication: y
postname: y
postattributetype: y
postdescription: y
postvaluestyle: y
posttype: y
```

```
postediting: y
poststorage: y
postenginename: y
poststatus: y
postassettypename: y
postupload: y
postexternalid: y
postexternalcolumn: y
postexternaltable: y
postcharsetname: y
```

## Sample Source File: Attribute

This is a sample source file for importing a product attribute named `footnotes` for the GE Lighting sample site. It works with the preceding sample configuration file.

```
<document>
<publication>GE Lighting</publication>
<name>footnotes</name>
<description>Footnotes</description>
<valuestyle>S</valuestyle>
<type>URL</type>
<editing>L</editing>
<storage>L</storage>
</document>
```

**Note**

Remember that all the dependencies and restrictions concerning the data type of a flex attribute apply whether you are creating an attribute through the Content Server interface (the **New** or **Edit** flex attribute forms) or through XMLPost. For information, read "Step 4: Create Flex Attributes" on page 317.

## Flex Definitions and Flex Parent Definitions

The flex definition and flex parent definition asset types are very similar and you code their configuration and source files in nearly the same way. They require several of the same tags in their source files and the same properties in their configuration files. Each has one additional property/tag.

The source file tags and configuration file properties for flex definitions and flex parent definitions are listed in the following table. Note that they are case-sensitive.

| Flex definition and flex parent definition tag and property | Description |
|---|---|
| tag:<br>`<internalname>`<br>property:<br>`postinternalname` | Required.<br>The name of the asset; this is a required value for all asset types. Flex definition and flex parent definition names are limited to 64 characters and they cannot contain spaces. |
| tag:<br>`<internaldescription>`<br>property:<br>`postinternaldescription` | Optional.<br>The description of the use or function of the asset. |
| tag:<br>`<renderid>`<br>property:<br>`postrenderid` | Optional. For flex definitions only.<br>The ID of the Template asset that is to be assigned to all the flex assets that are created with this flex definition. |
| tag:<br>`<parentselectstyle>`<br>property:<br>`postparentselectstyle` | Optional. For flex parent definitions only.<br>Defines how flex parents are to be selected when a user creates a flex asset using the definition.<br>This property/tag represents the **Parent Select Style** field in the **New** and **Edit** parent definition forms.<br>When using the tag in the source file, the options are `treepick` and `selectboxes`. |
| The next four tags and properties perform the same function as the buttons and fields in the **Product Parent Definition** section on the **New** and **Edit** forms for parent definitions and flex definitions. See "Step 5: (Optional) Create Flex Filter Assets" on page 321 and "Step 7: Create Flex Definition Assets" on page 324. | |
| tag:<br>`<OptionalSingleParentList>`<br>property:<br>`postOptionalSingleParentList` | Use this tag to specify any single optional parent definition. |

| Flex definition and flex parent definition tag and property | Description |
|---|---|
| tag:<br>`<RequiredSingleParentList>`<br>property:<br>`postRequiredSingleParentList` | Use this tag to specify any single required parent definition. |
| tag:<br>`<RequiredMultipleParentList>`<br>property:<br>`postRequiredMultipleParentList` | Use this tag to specify more than one required parent definitions. |
| tag:<br>`<OptionalMultipleParentList>`<br>property:<br>`postOptionalMultipleParentList` | Use this tag to specify more than one optional parent definition. |
| The next three tags and properties perform the same functions as the buttons and fields in the **Attributes** section on the **New** and **Edit** forms for flex definitions and flex parent definitions. See "Step 5: (Optional) Create Flex Filter Assets" on page 321 and "Step 7: Create Flex Definition Assets" on page 324. | |
| tag:<br>`<RequiredAttrList>`<br>property:<br>`postRequiredAttrList` | The list of attributes that are required for the flex parents or the flex assets that use the definition. |
| tag:<br>`<OptionalAttrList>`<br>property:<br>`postOptionalAttrList` | The list of attributes that are optional for the flex parents or the flex assets that use the definition. |
| tag:<br>`<OrderedAttrList>`<br>property:<br>`postOrderedAttrList` | The order in which all attributes, be they required or optional, should appear in the "New," "Edit," "Inspect," and similar forms.<br>If you use this tag, it replaces the other attribute tags. The example source file in this section shows an example of how to use this tag in a source file. |

A configuration file must include all the properties that could be used by any one of the assets of the type that the configuration file works with. The individual source files include only the tags that are needed to define those individual assets.

## Sample Configuration File: Flex Definition

The following example is a configuration file used to import product definitions for the GE Lighting sample site. It works with the sample source file immediately following this example.

```
xmlpost.url: http://izod19/servlet/ContentServer
xmlpost.numargs: 6
xmlpost.argname1: pagename
xmlpost.argvalue1: OpenMarket/Xcelerate/Actions/RemoteContentPost
xmlpost.argname2: AssetType
xmlpost.argvalue2: ProductTmpls
# Notice that this is the internal name of the asset type.
# The external name of this asset type is
# Product Definition.

xmlpost.argname3: authusername
xmlpost.argvalue3: user_editor
xmlpost.argname4: authpassword
xmlpost.argvalue4: user
xmlpost.argname5: inifile
xmlpost.argvalue5: futuretense.ini
xmlpost.argname6: startmenu
xmlpost.argvalue6: New Product Definition

xmlpost.success: Success
xmlpost.failure: Error
xmlpost.logfile: productdefpostlog.txt
xmlpost.deletefile: y

postpublication: y
postinternalname: y
postinternaldescription: y

postparentselectstyle: y

postOptionalSingleParentList: y
postRequiredSingleParentList: y
postRequiredMultipleParentList: y
postOptionalMultipleParentList: y

postRequiredAttrList: y
postOptionalAttrList: y
postOrderedAttrList: y

postrenderid: y
```

## Sample Source File: Flex Definition

The following source file, `lighting.xml`, is the source for a product definition named Lighting for the GE Lighting sample site. It works with the preceding sample configuration file.

```
<document>
<publication>GE Lighting</publication>
<internalname>Lighting</internalname>
<internaldescription>Generic Lighting Template</
internaldescription>
<RequiredAttrList>sku</RequiredAttrList>
<OptionalAttrList>
productdesc;caseqty;bulbshape;bulbsize;basetype;
colortemp;meanlength;lightcenterlength;reducedwattage;beamspread;
fixturetype;ballasttype;colorrenderingindex;minstarttemp;powerfact
or;
totalharmonicdist;spreadbeam10h;spreadbeam10v;spreadbeam50h;
spreadbeam50v;halogen;operatingposition;filamenttype;bulbimage;
baseimage;filamentimage;footnotes;price;life;voltage;wattage
</OptionalAttrList>
<parentselectstyle>treepick</parentselectstyle>
<OptionalMultipleParentList>SubCategory</
OptionalMultipleParentList>
</document>
```

Examine the list of attributes, above. When you include multiple values in a tag, separate them from each other with a semicolon (;).

Note that while GE Lighting uses the optional/multiple parent model, there are these other possible configurations:

```
<OptionalSingleParentList>flexparentdefinition
</OptionalSingleParentList>
<RequiredSingleParentList>flexparentdefinition
</RequiredSingleParentList>
<RequiredMultipleParentList>flexparentdefinition
</RequiredMultipleParentList>
```

### Supplying a List of Ordered Attributes

If you want to use the `<OrderedAttrList>` tag because the attributes need to be displayed in a specific order, do not also include the `<RequiredAttrList>` and `<OptionalAttrList>` tags. In the string contained in the `<OrderedAttrList>` tag, specify which attributes are required and which are optional, as follows:

- For required attributes, precede the attribute name with `R` (required)
- For optional attributes, precede the attribute name with or `O` (optional)
- Be sure to list the attributes in the desired order.
- Be sure to use a semicolon (;) to separate the values.

For example:

```
<OrderedAttrList>Rsku;Oproductdesc;Ocaseqty;Obulbshape;
Obulbsize;Obasetype;Ocolortemp;Omeanlength;Olightcenterlength;
Oreducedwattage;<OrderedAttrList>
```

## Flex Parents

You can use XMLPost to import flex parent assets in two ways:

- Individually. You code a separate XMLPost source file for each flex parent and an XMLPost configuration file that identifies the asset type and the pagename for the standard `RemoteContentPost` posting element. If you plan to use the BulkLoader utility, you must first import the flex parent assets with XMLPost in this way.
- As part of the flex family tree for a flex asset. If you are using XMLPost to import your flex assets (rather than the BulkLoader), you can combine the flex parents with the flex assets and import the flex parents as a part of a flex family tree, within the context of a specific flex asset. You code a separate XMLPost source file for each flex asset and identify all the parents for that flex asset in that source file. XMLPost then creates the variables for one flex asset and multiple flex parents (if they do not yet exist) when it parses the source file.

This section describes the source and configuration file for importing them individually. For information about importing them with the flex assets, see "Importing Flex Assets with XMLPost" on page 445.

### Sample Configuration File: Individual Flex Parent

The following example is a configuration file used to import product parents for the GE Lighting sample site. It works with the sample source file immediately following this example.

```
xmlpost.xmlfilenamefilter: .xml

xmlpost.url: http://izod19/servlet/ContentServer
xmlpost.numargs: 6
xmlpost.argname1: pagename
xmlpost.argvalue1: OpenMarket/Xcelerate/Actions/RemoteContentPost
xmlpost.argname2: AssetType
xmlpost.argvalue2: ProductGroups
# notice that you use the internal name of the asset type

xmlpost.argname3: authusername
xmlpost.argvalue3: user_editor
xmlpost.argname4: authpassword
xmlpost.argvalue4: user
xmlpost.argname5: inifile
xmlpost.argvalue5: futuretense.ini
xmlpost.argname6: startmenu
xmlpost.argvalue6: New Product Parent

xmlpost.success: Success
xmlpost.failure: Error
xmlpost.logfile: productdefpostlog.txt
xmlpost.deletefile: y

postpublication: y
postinternalname: y
postinternaldescription: y
postflexgrouptemplateid: y
```

```
postfgrouptemplatename: y
postParentList: y
postcat1: y
postcat2: y
```

### Sample Source File: Individual Flex Parent

The following source file creates a product parent (flex parent) named Halogen for the GE Lighting sample. It works with the preceding sample configuration file.

```
<document>
<publication>GE Lighting</publication>
<internalname>Halogen</internalname>
<fgrouptemplatename>Category</fgrouptemplatename>
<cat1>Halogen</cat1>
</document>
```

Remember that when you use the `RemoteContentPost` posting element, you must provide one source file for each parent asset.

## Importing Flex Assets with XMLPost

Before you can use XMLPost to import flex assets, you must have already imported the structural asset types (attributes, flex definitions, and flex parent definitions).

There are two posting elements that you can use for flex assets, `RemoteContentPost` or `addData`:

- The `addData` posting element creates parent assets for the flex asset if they do not yet exist. For example, if you are not using the BulkLoader utility, you use this posting element for the initial import of your flex assets.

  When you use the `addData` posting element, the source file must specify the entire family tree for the flex asset. You need a separate source file for each flex asset, but you can specify any number of parents for that flex asset in that source file and XMLPost creates the flex asset and its parents (if they do not yet exist).

- The `RemoteContentPost` element creates flex assets and sets values for their parents. Those parents must already exist. For example, if you plan to use BulkLoader once, just for the initial import and will use XMLPost from then on, you might want to use this posting element.

  When you use `RemoteContentPost` to import a flex asset, the source file must specify only the asset's immediate parents (which requires you to include fewer lines of code). However, if you need to create a new flex parent for the new flex asset, you must use the `addData` posting element and specify the entire family tree in the source file.

## Configuration File Properties and Source File Tags for Flex Assets

As with the structural asset types, you must use the internal name of the flex and flex parent asset types in your configuration and source files.

However, unlike the structural asset types, you do not need to include an argument for the asset type in the configuration file. Source files for flex assets have a required tag that identifies the asset type so you do not have to repeat this information in the configuration file.

### For the addData Posting Element

The following table lists the source file tags and configuration file properties for flex assets (and their flex parents) when you are using the `addData` posting element. Note that they are case-sensitive.

| Tag and property | Description |
|---|---|
| tag:<br>`<_ASSET_>`<br>property:<br>`post_ASSET_` | Required.<br>The internal name of the asset type. For example, the internal asset type names of the GE Lighting sample site flex assets are `Products`, `AArticles`, and `AImages`. |
| tag:<br>`<_TYPE_>`<br>property:<br>`post_TYPE_` | Required.<br>The name of the flex definition that this flex asset is using. |
| tag:<br>`<_ITEMNAME_>`<br>property:<br>`post_ITEMNAME_` | Required.<br>The name of the asset. |
| tag:<br>`<_ITEMDESCRIPTION_>`<br>property:<br>`post_ITEMDESCRIPTION_` | Optional.<br>The description of the asset. |

| Tag and property | Description |
|---|---|
| tag:<br>`<_GROUP_parentDefinitionName>`<br>property:<br>`post_GROUP_parentDefinitionName` | Optional.<br>The flex asset's parents. The configuration file must include a tag for each possible parent definition. For example, if your flex assets could have parents that use either of two parent definitions named Division and Department, the configuration file needs two properties that define a tag for each:<br>`post_Group_Department`<br>`post_Group_Division`<br>See "Specifying the Parents of a Flex Asset" on page 453 for more information about using this tag and property. |
| tag:<br>`<_GROUPDESCRIPTIONS_>`<br>property:<br>`post_GROUPDESCRIPTIONS_` | Optional.<br>If the parent that you are designating is new, you can also include the description of the parent definition. |
| tag:<br>`<displaytype>`<br>property:<br>`postdisplaytype` | Optional.<br>The name of the Template asset for the flex asset. |
| tag:<br>`<AttributeName>`<br>property:<br>`postAttributeName` | Include a property in the configuration file for each attribute that assets of the type can have (both required and optional). The source files then need to supply a value for each required attribute and any optional ones that apply to that asset.<br>For example, if there were an attribute named SKU, you would include a property called `postSKU` and in the source files, would include lines of code like this:<br>`<SKU>123445</SKU>.` |

## For the RemoteContentPost Posting Element

The following table lists the source file tags and configuration file properties for flex assets (and their flex parents) when you are using the `RemoteContentPost` posting element. Note that they are case-sensitive.

| Tag and property | Description |
|---|---|
| tag:<br>`<_DEFINITION_>`<br>property:<br>`post_DEFINITION_` | Required.<br>The name of the flex definition that this flex asset is using.<br>(Note that `post_TYPE` will also work.) |
| tag:<br>`<_ITEMNAME_>`<br>property:<br>`post_ITEMNAME_` | Required.<br>The name of the asset. |
| tag:<br>`<_ITEMDESCRIPTION_>`<br>property:<br>`post_ITEMDESCRIPTION_` | Optional.<br>The description of the asset. |
| `tag:`<br>`<ParentList>`<br>property:<br>`post_ParentList` | Optional.<br>The flex asset's immediate parents. |
| tag:<br>`<template>`<br>property:<br>`posttemplate` | Optional.<br>The name of the Template asset for the flex asset.<br>(Note that `postdisplaytype` will also work.) |
| tag:<br>`<AttributeName>`<br>property:<br>`postAttributeName` | Include a property in the configuration file for each attribute that assets of the type can have (both required and optional). The source files then need to supply a value for each required attribute and any optional ones that apply to that asset.<br>For example, if there were an attribute named SKU, you would include a property called `postSKU` and in the source files, would include lines of code like this:<br>`<SKU>123445</SKU>`. |

## Sample Flex Asset Configuration File for addData

This is a sample configuration file for the product asset type from the GE Lighting sample site. It invokes the `addData` posting element and works with the source file example immediately following this example file.

```
xmlpost.xmlfilenamefilter: .xml

#xmlpost.proxyhost: Future
#xmlpost.proxyport: 80

xmlpost.url: http://wally9:80/servlet/ContentServer

# notice that it uses addData
# rather than RemoteContentPost
xmlpost.numargs: 5

xmlpost.argname1: pagename
xmlpost.argvalue1: OpenMarket/Gator/XMLPost/addData

# Notice that you do not need to provide
# the name of the asset type because that information
# is required in the source files for flex assets.

xmlpost.argname2: inifile
xmlpost.argvalue2: futuretense.ini
xmlpost.argname3: authusername
xmlpost.argvalue3: editor
xmlpost.argname4: authpassword
xmlpost.argvalue4: xceleditor
xmlpost.argname5: startmenu
xmlpost.argvalue5: New Product

xmlpost.success: Success
xmlpost.failure: Error
xmlpost.logfile: productdatalog.txt

xmlpost.postdeletefile: y

post_ASSET_: y
post_ITEMNAME_: y
post_TYPE_: y
post_GROUP_Category: y
post_GROUP_SubCategory: y
postpublication: y
postsku: y
postproductdesc: y
postcaseqty: y
postbulbshape: y
postbulbsize: y
postbasetype: y
postcolortemp: y
postmeanlength: y
```

```
postlightcenterlength: y
postreducedwattage: y
postbeamspread: y
postfixturetype: y
postballasttype: y
postcolorrenderingindex: y
postminstarttemp: y
postpowerfactor: y
posttotalharmonicdist: y
postspreadbeam10h: y
postspreadbeam10v: y
postspreadbeam50h: y
postspreadbeam50v: y
posthalogen: y
postoperatingposition: y
postfilamenttype: y
postbulbimage: y
postbaseimage: y
postfilamentimage: y
postfootnotes: y
postcat1: y
postcat2: y
postprice: y
postvoltage: y
postwattage: y
postlife: y
```

## Configuration File Properties and Attributes of Type Blob (or URL)

If the asset type has an attribute of type `blob` (or `url`), the configuration file needs two entries for the tag that references the attribute: one to identify the attribute and one to identify the file name of either a) the file that holds the content for the attribute (an upload field) or b) the name that you want Content Server to give the file that it creates from text entered directly into a text field (a text field of type `blob` or `URL`).

### Attribute of Type Blob (or URL) as an Upload Field

There are no attributes of type blob in the GE Lighting sample site. However, lets say that there is an attribute of type `blob` named `footnotes`. It is an upload field with a **Browse** button for finding the file rather than a text field that you enter text in to. Therefore it has two properties:

- `posttag`, which in this scenario is `postfootnotes: y`
- `filetag`, which in this scenario is `filefootnotes: y`

When you include a value for this attribute in a source file, you use the following convention:

```
<footnotes>FileName.txt</footnotes>
```

Note that when you are importing an asset that has this kind of field (attribute), the file that holds the text that you want to store as the attribute value for the flex asset must be located in the same directory as the source file for the asset.

### Attribute of Type Blob (or URL) as a Text Field

If the fictitious `footnotes` attribute is a field that takes text directly rather than a file, the configuration file requires the following properties:

- `postfootnotes: y`
- `postfootnotes_file: y`

Then, when you include a value for the attribute in the source file, you use the following convention:

```
<footnotes>lots and lots of text</footnotes>
<footnotes_file>FileNameYouWantUsed.txt</footnotes_file>
```

## Sample Flex Asset Source File for addData

This following source file works with the example flex asset configuration file preceding this section. This source file creates a lightbulb product named `10004` from the product definition named `Lighting`:

```
<document>

# the first three tags are required
<_ASSET_>Products</_ASSET_>
<_ITEMNAME_>10004</_ITEMNAME_>
<_TYPE_>Lighting</_TYPE_>

# This tag is required because the publication is
# not set in the configuration file
<publication>GE Lighting</publication>

# This tag assigns a Template asset to the product
<displaytype>Lighting Detail</displaytype>

# The rest of these tags set flex attribute values for the product
<price>5</price>
<sku>10004</sku>
<productdesc>F4T5/CW</productdesc>
<caseqty>24</caseqty>
<bulbshape>T</bulbshape>
<bulbsize>5</bulbsize>
<basetype>Miniature Bipin (G5)</basetype>
<colortemp>4100</colortemp>
<meanlength></meanlength>
<lightcenterlength></lightcenterlength>
<reducedwattage></reducedwattage>
<beamspread></beamspread>
<fixturetype></fixturetype>
<ballasttype></ballasttype>
<colorrenderingindex>60</colorrenderingindex>
<minstarttemp></minstarttemp>
<powerfactor></powerfactor>
<totalharmonicdist></totalharmonicdist>
<spreadbeam10h></spreadbeam10h>
<spreadbeam10v></spreadbeam10v>
```

```
<spreadbeam50h></spreadbeam50h>
<spreadbeam50v></spreadbeam50v>
<halogen></halogen>
<operatingposition></operatingposition>
<filamenttype></filamenttype>
<bulbimage>BLB-260.gif</bulbimage>
<baseimage>BLB-250.gif</baseimage>
<filamentimage></filamentimage>
<footnotes>
</footnotes>
<life>6000</life>
<voltage></voltage>
<wattage>4</wattage>
<cat1>Fluorescent</cat1>
<cat2>Preheat Lamps</cat2>

<!-- GROUP tags that specify the parents. Remember that you have
to
specify the entire family tree for the flex asset when using the
addData posting element-->

<_GROUP_Category>Fluorescent</_GROUP_Category>
<_GROUP_SubCategory>Preheat Lamps</_GROUP_SubCategory>
</document>
```

The preceding source file set the product's parent to Preheat Lamps and the parent of Preheat Lamps to Fluorescent.

## Handling Special Characters

XMLPost uses the HTTP POST protocol, which means that it sends data in an HTTP stream. Therefore, certain characters are considered to be special characters and must be encoded because they are included in URLs.

If your source file includes attribute values that contains any of the special characters listed in the following table, be sure to replace all instances of that character with its corresponding URL encoding sequence, found in the Values for Special Characters section of this guide.

## Flex Assets and Their Parents

The `GROUP` tags specify the parents in the family tree. When XMLPost uses the `addData` posting element and parses the `GROUP` section of the source file, it does the following:

1. Determines which parent definitions are legal for an asset using this flex definition.
2. For each legal parent definition, it verifies whether the source file specifies a parent of that definition:
   - If yes, it sets the parent and if the parent does not yet exist, it creates the parent.
   - If no, it does not set the parent. However, if a parent of that definition is required, it returns an error.

## Specifying the Parents of a Flex Asset

To specify the parents of a flex asset, you provide the name of the parents nested in the `<_GROUP_parentDefinitionName>` tag. For example:

```
<_GROUP_subcategory>Blacklights</_GROUP_subcategory>
```

Where `subcategory` is the name of the parent definition for the Blacklights parent (product parent).

Remember that you must specify the entire family tree for the flex asset. In the GE Lighting sample site, a product asset has a parent and a grandparent. In addition to specifying the parent for the lightbulb (Blacklight), you need to specify the grandparent. For example:

```
<_GROUP_subcategory>Blacklights</_GROUP_subcategory>
<_GROUP_category>Fluorescent</_GROUP_category>
```

## Setting Attribute Values for Parents

How does XMLPost know which parent the attribute values belong to? It does not. If an attribute can belong to more than one parent, you must specify which parent it belongs to. For example, let's say that the bulbshape attribute is assigned to parents rather than products. In this case, you would include a line of code such as this:

```
<bulbshape>Halogen=T</bulbshape>
```

## Setting Multiple Values in a Flex Source File

All of the tags that configure parents and the tags that specify attributes (as long as the attribute is configured to accept multiple values) can handle multiple values. Those tags are as follows:

- `_GROUP_parentDefinitionName`
- `_GROUPDESCRIPTIONS_`
- the attribute tags

When you have multiple parents from the same definition for a flex asset, you provide all of the names of the parents in the same `_GROUP_parentDefinitionName` tag and you use a semicolon (;) to separate the parent names.

For example:

```
<_GROUP_Cateogry>Incandescent;Halogen</_GROUP_Category>
```

When XMLPost imports this asset, it sets its parents as Incandescent and Halogen, which are both of the Category parent definition. If Incandescent and Halogen do not exist yet, XMLPost creates them.

You use a similar syntax when you want to set multiple attribute values for the multiple parents. Once again, let's say that the Category definition requires that parents of that definition have a value for the bulbshape attribute. You can set the value of the bulbshape attribute for both of the parents that were specified by the `<_GROUP_Category>` tag as follows:

```
<bulbshape>Incandescent=E;K:Halogen=T</bulbshape>
```

Note the following about this syntax:

- You use `parentName=attributeValue` pairs to set the attribute value (`Halogen=T`).

- You use a colon to separate the parents from each other (`Incandescent=S:Halogen=T`).
- You use a semicolon to separate the attribute values for a parent when that parent has more than one value for the attribute (`Incandescent=E;K:Halogen=T`).

And, as mentioned, you can specify descriptions for the parents that you identify in the same tag, too. For example:

```
<_GROUPDESCRIPTIONS>
Incandescent=From Detroit:Halogen=From Chicago
</_GROUPDESCRIPTIONS>
```

## Sample Flex Asset Configuration File for RemoteContentPost

This is a sample configuration file for the product asset type from the GE Lighting sample site. It works with the source file example immediately following this example file.

```
xmlpost.xmlfilenamefilter: .xml

#xmlpost.proxyhost: Future
#xmlpost.proxyport: 80

xmlpost.url: http://wally9:80/servlet/ContentServer
xmlpost.numargs: 5

xmlpost.argname1: pagename
xmlpost.argvalue1: OpenMarket/Xcelerate/Actions/RemoteContentPost

# Notice that you do not need to provide
# the name of the asset type because that information
# is required in the source files for flex assets.

xmlpost.argname2: inifile
xmlpost.argvalue2: futuretense.ini
xmlpost.argname3: authusername
xmlpost.argvalue3: editor
xmlpost.argname4: authpassword
xmlpost.argvalue4: xceleditor
xmlpost.argname5: startmenu
xmlpost.argvalue5: New Product

xmlpost.success: Success
xmlpost.failure: Error
xmlpost.logfile: productdatalog.txt

xmlpost.postdeletefile: y

postpublication: y

post_ASSET_: y
post_ITEMNAME_: y
post_DEFINITION_: y
posttemplate: y
```

```
postsku: y
postproductdesc: y
postcaseqty: y
postbulbshape: y
postbulbsize: y
postbasetype: y
postcolortemp: y
postmeanlength: y
postlightcenterlength: y
postreducedwattage: y
postbeamspread: y
postfixturetype: y
postballasttype: y
postcolorrenderingindex: y
postminstarttemp: y
postpowerfactor: y
posttotalharmonicdist: y
postspreadbeam10h: y
postspreadbeam10v: y
postspreadbeam50h: y
postspreadbeam50v: y
posthalogen: y
postoperatingposition: y
postfilamenttype: y
postbulbimage: y
postbaseimage: y
postfilamentimage: y
postfootnotes: y
postcat1: y
postcat2: y
postprice: y
postvoltage: y
postwattage: y
postlife: y

postParentList: y
```

## Sample Flex Asset Source File for RemoteContentPost

This following source file works with the example configuration file immediately preceding this section. This source file creates a lightbulb product named `10004` from the product definition named `Lighting`:

```
<document>

# the first three tags are required
<_ASSET_>Products</_ASSET_>
<_ITEMNAME_>10004</_ITEMNAME_>
<_DEFINITION_>Lighting</_DEFINITION_>

# This tag is required because the publication is
# not set in the configuration file
<publication>GE Lighting</publication>
```

```
# This tag assigns a Template asset to the product
<template>Lighting_Detail</template>

# The rest of these tags set flex attribute values for the product
<price>5</price>
<sku>10004</sku>
<productdesc>F4T5/CW</productdesc>
<caseqty>24</caseqty>
<bulbshape>T</bulbshape>
<bulbsize>5</bulbsize>
<basetype>Miniature Bipin (G5)</basetype>
<colortemp>4100</colortemp>
<colorrenderingindex>60</colorrenderingindex>
<bulbimage>BLB-260.gif</bulbimage>
<baseimage>BLB-250.gif</baseimage>
<filamentimage></filamentimage>
<life>6000</life>
<voltage></voltage>
<wattage>4</wattage>
<cat1>Fluorescent</cat1>
<cat2>Preheat Lamps</cat2>

# this tag sets the immediate parents only
<ParentList>Preheat Lamps</ParentList>

</document>
```

The preceding source file sets several attribute values for the product and sets its immediate parent to Preheat Lamps. This parent must already exist.

# Editing Flex Assets with XMLPost

You can edit the following information for flex assets and flex parent assets with XMLPost:

- The value of an attribute
- The asset's parents (either the flex asset's parents or the parent's parents)

You cannot edit attribute assets, flex definition assets, or flex parent definition assets with XMLPost.

To edit the attribute value for a flex asset, the source file needs to include only the name of the asset and the attribute that you want to change.

To edit the attribute value for a flex parent, you must provide the context of a flex asset. The source file must name the flex asset and can then reference just parent and the parent attribute that you want to change. But you must specify a flex asset for XMLPost to start with so that it can work its way through the family tree.

## Configuration Files for Editing Flex Assets

There are two differences in the configuration file for editing a flex asset: the pagename argument and an additional tag and property.

### Pagename Argument

The `pagename` argument must be set to: `OpenMarket/Gator/XMLPost/modifyData`.

For example:

```
xmlpost.argname1: pagename
xmlpost.argvalue1: OpenMarket/Gator/XMLPost/modifyData
```

You invoke XMLPost from the command line as usual, identifying the configuration file and the source files.

### Additional Tag/Property

You can use the following optional tag and property when you are editing a flex asset:

- tag: `<_REMOVE_parentDefinitionName>`
- property: `post_REMOVE_parentDefinitionName`

It removes a parent from the flex asset.

## Source Files for Editing Flex Assets

The source file for an edited flex asset does not need to include all the information for that asset—you only need to provide the information that you want to change. Any attributes that you do not specify are not modified in any way.

### Changing the Value of an Attribute

To change the value of an attribute, you specify the new attribute value in the source file. When XMLPost runs the import, it writes over the old value with the value provided in the source file.

The following sample source file changes two attribute values (bulbshape and bulbsize) for the GE Lighting product named 10004 that was defined in "Sample Flex Asset Source File for addData" on page 451.

```
<document>
<!-- predefined xml tags (required) -->
   <_ASSET_>Products</_ASSET_>
   <_ITEMNAME_>10004</_ITEMNAME_>
   <_TYPE_>Lighting</_TYPE_>

<!-- attribute xml tags -->
   <bulbshape>E</bulbshape>
   <bulbsize>9</bulbsize>
</document>
```

## Removing an Attribute Value

To remove an attribute value and leave it blank, code a line that names the attribute and specify `_EMPTY_` as the attribute's value.

For example:

```
<bulbsize>_EMPTY_</bulbsize>
```

You can also edit attribute values for parents. Let's say that the bulbsize attribute is set at the parent level. If that were the case, the following lines of code would set two parents and provide a value for bulbsize for each:

```
<_GROUP_SubCategory>All-Weather Lamps;Appliance Lamps
</GROUP_SubCategory>
<bulbsize>All-Weather Lamps=10:Appliance Lamps=8</bulbsize>
```

### Option 1

This line of code clears the bulbsize for the All-Weather Lamps parent:

```
<bulbsize>All-Weather Lamps=_EMPTY_:Appliance Lamps=8</bulbsize>
```

### Option 2

Alternatively, you could just use this line of code, without repeating the value for Appliance Lamps:

```
<bulbsize>All-Weather Lamps=_EMPTY_</bulbsize>
```

## Editing Parent Relationships

You can use XMLPost to make the following edits to the parent relationships for a flex asset:

- Add another parent to the existing parents.
- Change a parent from one parent to another.

The `GROUP_parentDefinitionName` tag works differently than do the attribute tags.

- When you use an attribute tag, XMLPost writes the new value over the old value.
- When you use a `GROUP_parentDefinitionName` tag, XMLPost does not overwrite an old parent with a new parent, even when the parent definition name is the same—it adds the new parent to the list of parents that the asset has, which may not be what you want.

To add another parent to the list of existing parents, include the line of code in the source file. For example:

```
<_GROUP_SubCategory>Blacklights</_GROUP_SubCategory>
```

If you want to remove a parent, use the <_REMOVE _> tag. Note that you must be careful not to remove a required parent unless you are replacing it.

```
<_REMOVE_Processor>Appliance Lamps</_REMOVE_Processor>
```

# Deleting Assets with XMLPost

If you have CS-Direct Advantage, you can use XMLPost to delete any asset of any type. There are two requirements:

- Your configuration file must instruct XMLPost to call the `deleteData` element.

  For example:

  ```
  xmlpost.argname1: pagename
  xmlpost.argvalue1: OpenMarket/Gator/XMLPost/deleteData
  ```

- There are two required source file tags and configuration file properties:

  `<_ASSET_>/post_ASSET_` which identifies the asset type of the asset you want to delete.

  `<_ITEMNAME>/post_ITEMNAME_` which identifies the asset you want to delete.

When XMLPost uses this posting element, it changes the value in the `Status` column for that asset to `VO` for void. (It does not physically remove it from the database).

## Configuration Files for Deleting Assets

Here is an example configuration file:

```
xmlpost.xmlfilenamefilter: .xml

xmlpost.url: http://izod19/servlet/ContentServer
xmlpost.numargs: 4
xmlpost.argname1: pagename
xmlpost.argvalue1: OpenMarket/Gator/XMLPost/deleteData
xmlpost.argname2: authusername
xmlpost.argvalue2: user_editor
xmlpost.argname3: authpassword
xmlpost.argvalue3: user
xmlpost.argname4: inifile
xmlpost.argvalue4: futuretense.ini

xmlpost.success: Success
xmlpost.failure: Error
xmlpost.logfile: productdefpostlog.txt
xmlpost.deletefile: y

postpublication: y
post_ASSET_: y
post_ITEMNAME_: y
```

You invoke XMLPost from the command line as usual.

## Source Files for Deleting Assets

The source files for deleting assets are short and simple. For example:

```
<document>
   <_ASSET_>Products</_ASSET_>
   <_ITEMNAME_>Pentium 90</_ITEMNAME_>
   <publication>my publication</publication>
</document>
```

This code instructs XMLPost to delete a product asset named `Pentium 90` (it changes the status of `Pentium 90` to `VO`, for void).

Chapter 22

# Importing Flex Assets with the BulkLoader Utility

This chapter describes the BulkLoader utility, which you use to import flex assets during the during the initial setup of your Content Server system.

It contains the following sections:


- Overview of BulkLoader
- Importing Flex Assets from Flat Tables
- Importing Flex Assets Using a Custom Extraction Mechanism
- Approving Flex Assets with the BulkApprover Utility

# Overview of BulkLoader

The BulkLoader utility enables you to quickly extract large amounts of flex asset data in a user-defined way from your own data sources and import that data into the Content Server database on any of your systems (development, management, testing, or delivery).

The extraction mechanism is abstracted away using a Java interface that customers can implement. BulkLoader invokes methods on this interface to extract input data from your data sources. For backward functional and data compatibility, Content Server also includes an implementation of this Java interface so that BulkLoader will still be able to extract data from an external JDBC-compliant data source.

## BulkLoader Features

Features in BulkLoader include the following:

- Support for a user-defined extraction mechanism, using a Java API. Users can provide a custom implementation of this extraction interface or use the built-in support for extracting from a JDBC data source.
- Support for inserts, voids, and updates of flex asset and group data.
- Support for incremental inserts, voids and updates.
- Performance improvements for higher throughput, using concurrent multi-threaded import operations while data extraction is in progress.
- Support for chunk (slice) processing of input data.
- Support for importing asset data that belongs to multiple flex families.
- Backward functional and data compatibility. Supports importing asset data from an external JDBC source.

## How BulkLoader Works

The BulkLoader has been redesigned for higher performance, throughput, and scalability. Instead of reading all input data and then generating output SQL files, BulkLoader reads input data in chunks. As soon as each chunk is read, it is handed over to an import thread while the main BulkLoader thread goes back to read the next chunk. The import thread uses a direct JDBC connection to the Content Server database. In this way, reading and importing are done in parallel, thereby achieving higher throughput. For scalability, users can increase the number of BulkLoader import threads if the database machine's hardware has additional CPUs and an I/O configuration that supports higher concurrency.

The BulkLoader utility requires a configuration file containing parameters that specify the number of processing threads, the name of the Java class that implements the data extraction interface, commit frequency, the starting unique ID to be used as the asset ID, and more.

The following diagrams show a client-specific implementation and the built-in "out of the box" implementation supplied by FatWire:

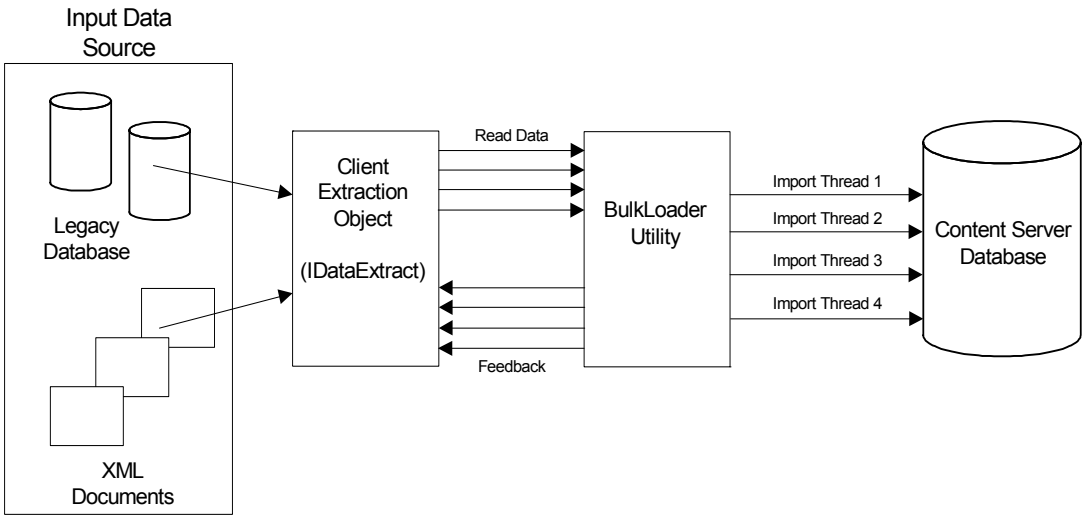



**Figure 8:** Client-specific implementation of BulkLoader

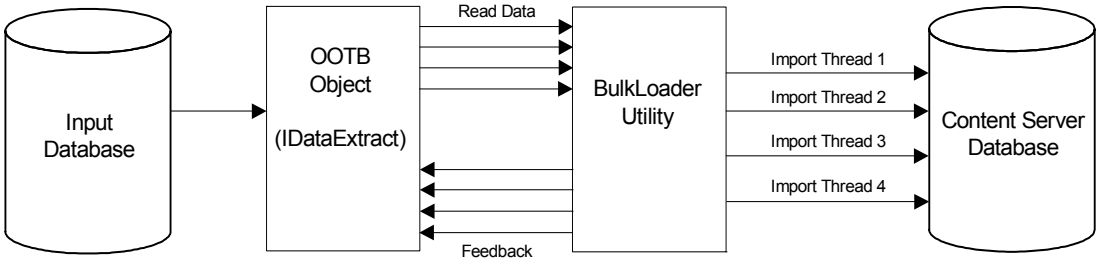



**Figure 9:** Built-in OOTB (“out of the box”) implementation of BulkLoader

## Using the BulkLoader Utility

There are two ways to use the BulkLoader depending on how you supply input data to import into the Content Server database.

- If you want BulkLoader to import input data from an external JDBC data source, you provide input data in a flat table or view.
- If you want to provide your own way of supplying input data to the BulkLoader, you use a Java object that implements the extraction interface, IDataExtract.

**Note**

For reference, sample BulkLoader code is provided on the Content Server installation medium, in the “Samples” folder. The same folder contains the `readme.txt` file that describes the sample files.

# Importing Flex Assets from Flat Tables

This section describes the general procedure that you use to import flex assets with BulkLoader, followed by subsequent sections that describe each step in detail. Using this model, you can import new flex assets and parents, as well as void assets that were previously imported. This model also supports changing and deleting attribute values for existing assets.

## The Basic Steps

The basic process of importing flex assets with the BulkLoader utility is as follows:

1. Use XMLPost to import the structural assets into the Content Server database on the management system. The structural flex assets are as follows: attribute editors, flex attributes, flex parent definitions, flex definitions, and flex parent assets.
2. Write a view or a stored procedure that gives you a view of the source database that you want to import into the Content Server database as a flat table. This flat table is your source table.
3. In the same source database, create a mapping table with two columns: one column that lists the names of the columns in the source file and the other column that lists the names that are used for those attributes in the Content Server database.
4. Code a configuration file that identifies the source table and the mapping table.
5. Put the configuration file on a system from which you have access to both the Content Server database on the management system, and to your source database.
6. Stop the application server on the management system.
7. Run the BulkLoader utility. BulkLoader will import the flex asset data and gives the feedback in a table named `bulk_feedback`, that has been created at the input data source.
8. Restart the application server on the management system.
9. Use the BulkApprover utility to approve all of the assets that were loaded

**Note**

Because the BulkLoader utility is designed for speed, it does **not** check for the existence of the attributes or flex parent definitions or flex definitions. You must import all of the structural asset types before you run the BulkLoader utility.

## Driver Requirements

The BulkLoader requires JDBC (or Java DataBase Connectivity) drivers, which are not provided by FatWire Corporation You must obtain JDBC drivers for both the source database and the destination database, that is, the Content Server database, even if your Content Server database is MS SQL2000.

If you have a source database that is ODBC-compliant, you can use a JDBC-ODBC bridge, which is included as part of the Java SDK from Sun. For MS SQL2000, however, that is not recommended—use JNet Direct, instead.

### Requirement for DB2

If you are using the DB2 database with your Content Server system, you must run the `usejdbc2.bat` file on the client machine before you can use BulkLoader. You only need to run the batch file once; then you can run BulkLoader as usual.

## Step 1: Use XMLPost to Import Structural Assets

Use XMLPost and the `RemoteContentPost` posting element to import the structural assets into the Content Server database on the management system. Import assets of the following types:

- attribute editors
- flex attributes
- flex parent definitions
- flex definitions
- flex parent assets

For information about this step, see "Importing the Structural Asset Types in the Flex Model" on page 434.

## Step 2: Create the Input Table (Data Source)

You must create input flat tables (data sources) for holding all new asset data and for holding update data. These are flat tables/views in which each row corresponds to a single CS-Direct Advantage flex asset item and each column corresponds to a CS-Direct Advantage flex attribute asset for the BulkLoader utility.

There is no requirement regarding the names of columns in the data source, but you must supply a separate mapping table, described in "Step 3: Create the Mapping Table" on page 467.

### Inserts

The name of the data source table is specified by the `inputTable` parameter in the configuration file.

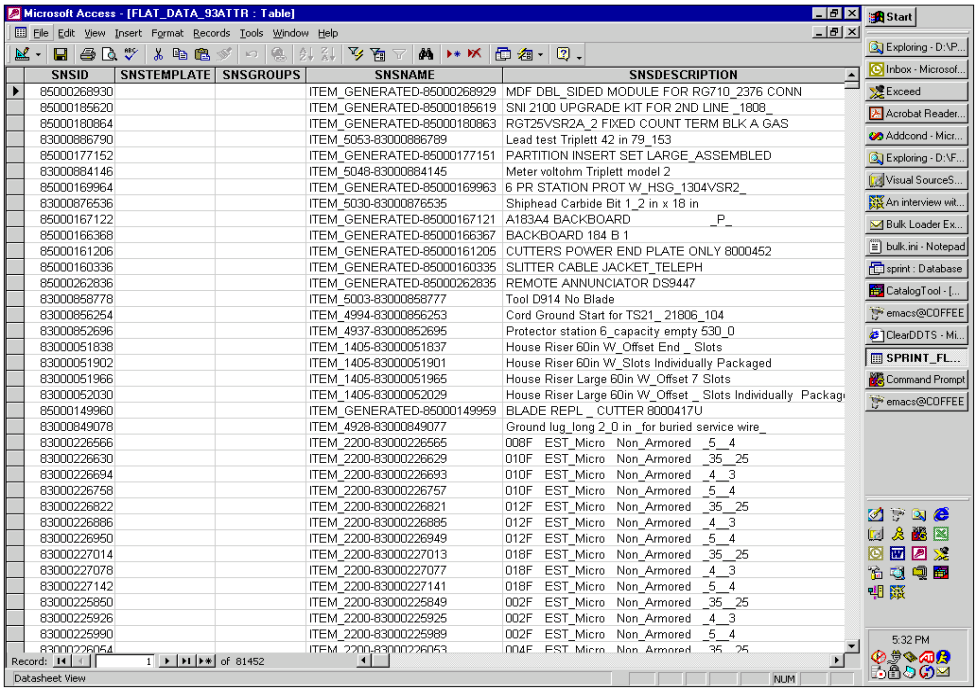
The source table must also include the names of the following four columns, which you specify in the configuration file with the following properties:

- `inputTableTemplateColumn` – The name of the column in the source table that holds the names of the flex definitions.
- `inputTableNameColumn` – The name of the column in the source table that holds the names of the flex assets. The name of this column cannot exceed 64 characters.
- `inputTableDescriptionColumn` – The name of the column in the source table that holds the description of the flex assets.
- `inputTableGroupsColumn` – The name of the column in the source table that holds the names of the parent definitions. Each value in this column can include multiple flex parent definition names, separated by the `multivalueDelimeter` character, which is defined in the configuration file.

Note that you can optionally specify the name of the column that serves as a unique identifier for each input item, using the following parameter in the configuration file: `inputTableUniqueIdColumn`. If there is no value assigned for this parameter,

BulkLoader will generate a unique identifier for each input item and store it in a mapping table (`bulkloader_ids`) in the Content Server database.

This is an example of a source table (input table):



| SNSID | SNSTEMPLATE | SNSGROUPS | SNSNAME | SNSDESCRIPTION |
|---|---|---|---|---|
| 85000268930 | | | ITEM_GENERATED-85000268929 | MDF DBL_SIDED MODULE FOR RG710_2376 CONN |
| 85000185620 | | | ITEM_GENERATED-85000185619 | SNI 2100 UPGRADE KIT FOR 2ND LINE _1808_ |
| 85000180864 | | | ITEM_GENERATED-85000180863 | RGT25VSR2A_2 FIXED COUNT TERM BLK A GAS |
| 83000886790 | | | ITEM_5053-83000886789 | Lead test Triplett 42 in 79_153 |
| 85000177152 | | | ITEM_GENERATED-85000177151 | PARTITION INSERT SET LARGE_ASSEMBLED |
| 83000884146 | | | ITEM_5048-83000884145 | Meter voltohm Triplett model 2 |
| 85000169964 | | | ITEM_GENERATED-85000169963 | 6 PR STATION PROT W_HSG_1304VSR2_ |
| 83000876536 | | | ITEM_5030-83000876535 | Shiphead Carbide Bit 1_2 in x 18 in |
| 85000167122 | | | ITEM_GENERATED-85000167121 | A183A4 BACKBOARD _P_ |
| 85000166368 | | | ITEM_GENERATED-85000166367 | BACKBOARD 184 B 1 |
| 85000161206 | | | ITEM_GENERATED-85000161205 | CUTTERS POWER END PLATE ONLY 8000452 |
| 85000160336 | | | ITEM_GENERATED-85000160335 | SLITTER CABLE JACKET_TELEPH |
| 85000262836 | | | ITEM_GENERATED-85000262835 | REMOTE ANNUNCIATOR DS9447 |
| 83000858778 | | | ITEM_5003-83000858777 | Tool D914 No Blade |
| 83000856254 | | | ITEM_4994-83000856253 | Cord Ground Start for TS21_ 21806_104 |
| 83000852696 | | | ITEM_4937-83000852695 | Protector station 6_capacity empty 530_0 |
| 83000051838 | | | ITEM_1405-83000051837 | House Riser 60in W_Offset End _ Slots |
| 83000051902 | | | ITEM_1405-83000051901 | House Riser 60in W_Slots Individually Packaged |
| 83000051966 | | | ITEM_1405-83000051965 | House Riser Large 60in W_Offset 7 Slots |
| 83000052030 | | | ITEM_1405-83000052029 | House Riser Large 60in W_Offset _ Slots Individually Packag |
| 85000149960 | | | ITEM_GENERATED-85000149959 | BLADE REPL _ CUTTER 8000417U |
| 83000849078 | | | ITEM_4928-83000849077 | Ground lug_long 2_0 in _for buried service wire_ |
| 83000226566 | | | ITEM_2200-83000226565 | 008F EST_Micro Non_Armored _5__4 |
| 83000226630 | | | ITEM_2200-83000226629 | 010F EST_Micro Non_Armored _35__25 |
| 83000226694 | | | ITEM_2200-83000226693 | 010F EST_Micro Non_Armored _4__3 |
| 83000226758 | | | ITEM_2200-83000226757 | 010F EST_Micro Non_Armored _5__4 |
| 83000226822 | | | ITEM_2200-83000226821 | 012F EST_Micro Non_Armored _35__25 |
| 83000226886 | | | ITEM_2200-83000226885 | 012F EST_Micro Non_Armored _4__3 |
| 83000226950 | | | ITEM_2200-83000226949 | 012F EST_Micro Non_Armored _5__4 |
| 83000227014 | | | ITEM_2200-83000227013 | 018F EST_Micro Non_Armored _35__25 |
| 83000227078 | | | ITEM_2200-83000227077 | 018F EST_Micro Non_Armored _4__3 |
| 83000227142 | | | ITEM_2200-83000227141 | 018F EST_Micro Non_Armored _5__4 |
| 83000225850 | | | ITEM_2200-83000225849 | 002F EST_Micro Non_Armored _35__25 |
| 83000225926 | | | ITEM_2200-83000225925 | 002F EST_Micro Non_Armored _4__3 |
| 83000225990 | | | ITEM_2200-83000225989 | 002F EST_Micro Non_Armored _5__4 |
| 83000226054 | | | ITEM_2200-83000226053 | 004F EST_Micro Non_Armored _35__25 |

Based on the column names in this source table, the source table properties in the corresponding configuration file would be set as follows:

```
inputTableTemplateColumn=SNSTEMPLATE
inputTableNameColumn=SNSNAME
inputTableDescriptionColumn=SNSDESCRIPTION
inputTableGroupsColumn=SNSGROUPS
```

## Updates

If you want to update attribute data for existing assets, to add new parents or delete existing parents for existing assets, then you need to use the update parameter.

Use the `inputTableForUpdates` parameter in the configuration file to specify the name of the data source table. The source table must also include the names of the following three columns, which you specify in the configuration file with the following properties:

- `inputTableForUpdatesUniqueIdColumn` – The name of the column in the source table that uniquely identifies the flex asset or parent in the Content Server database.
- `inputTableForUpdatesDeleteGroupsColumn` – The name of the column in the source table that specifies a list of parents to be deleted for the current flex asset.
- `inputTableForUpdatesAddGroupsColumn` – The name of the column in the source table that specifies a list of parents to be added for the current flex asset.

BulkLoader interprets column values as follows when applying updates to the attributes:

- A null value in a specific attribute column indicates that the attribute for the current flex asset should be deleted. For example, a null value in the `deletegroups` column indicates that no parents need to be deleted. A null value in the `addgroups` column indicates that no parents need to be added.
- A non-null value indicates that the existing attribute value should be replaced with the given value. For example, a non-null value in the `deletegroups` column specifies a list of parents to be deleted. A non-null value for `addgroups` denotes the addition of new parents to a given flex asset.

## Step 3: Create the Mapping Table

You must also create a mapping table for the BulkLoader utility, and it must have the following two columns:

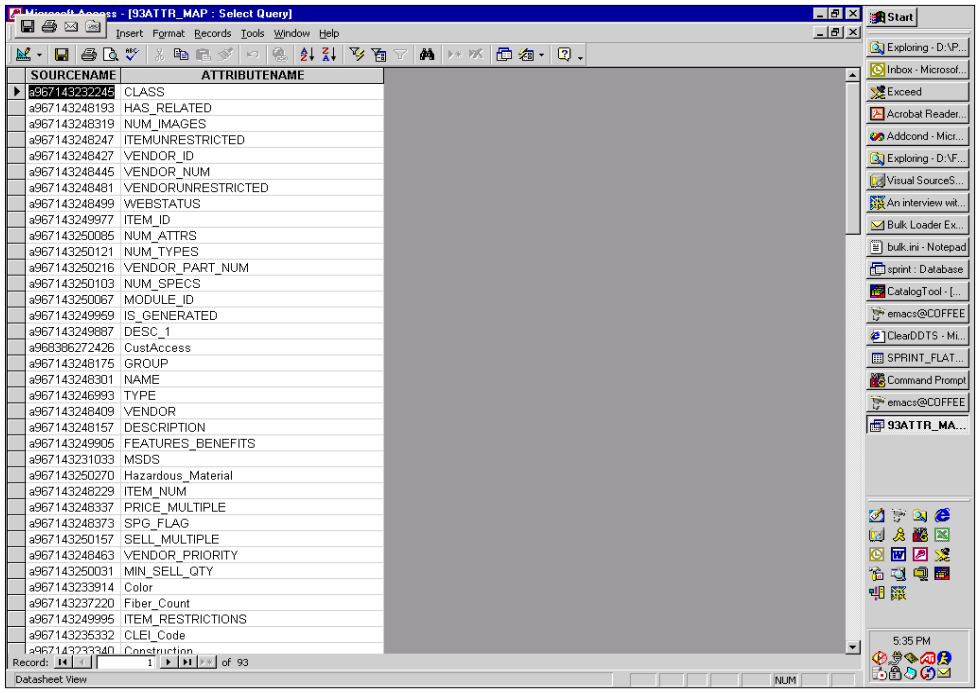
- A column that holds the names of the flex attribute columns in your flat data source
- A column that holds their corresponding names in the Content Server database

The mapping table provides a one-to-one correspondence between these two columns. For example, your source table might have a column of vendor names with an automatically generated name like `A96714328445` that maps to a CS-Direct Advantage product attribute asset named, simply, `VENDOR_ID`.

You include the following configuration file properties for the mapping table:

- `inputAttributeMapTable` – the name of the mapping table file
- `inputAttributeMapTableKeyCol` – the name of the column in the mapping table that lists the attribute names in the source table
- `inputAttributeMapTableValCol` – the name of the column that lists the corresponding attribute asset names in the Content Server database

The following is an example of a mapping table:



| SOURCENAME | ATTRIBUTENAME |
|---|---|
| a967143232245 | CLASS |
| a967143248193 | HAS_RELATED |
| a967143248319 | NUM_IMAGES |
| a967143248247 | ITEMUNRESTRICTED |
| a967143248427 | VENDOR_ID |
| a967143248445 | VENDOR_NUM |
| a967143248481 | VENDORUNRESTRICTED |
| a967143248499 | WEBSTATUS |
| a967143249977 | ITEM_ID |
| a967143250085 | NUM_ATTRS |
| a967143250121 | NUM_TYPES |
| a967143250216 | VENDOR_PART_NUM |
| a967143250103 | NUM_SPECS |
| a967143250067 | MODULE_ID |
| a967143249959 | IS_GENERATED |
| a967143249887 | DESC_1 |
| a968386272426 | CustAccess |
| a967143248175 | GROUP |
| a967143248301 | NAME |
| a967143246993 | TYPE |
| a967143248409 | VENDOR |
| a967143248157 | DESCRIPTION |
| a967143249905 | FEATURES_BENEFITS |
| a967143231033 | MSDS |
| a967143250270 | Hazardous_Material |
| a967143248229 | ITEM_NUM |
| a967143248337 | PRICE_MULTIPLE |
| a967143248373 | SPG_FLAG |
| a967143250157 | SELL_MULTIPLE |
| a967143248463 | VENDOR_PRIORITY |
| a967143250031 | MIN_SELL_QTY |
| a967143233914 | Color |
| a967143237220 | Fiber_Count |
| a967143249996 | ITEM_RESTRICTIONS |
| a967143235332 | CLEI_Code |
| a967143233340 | Construction |

Based on the column names in this source table, the source table properties in the corresponding configuration file would be set as follows:

```
inputAttributeMapTable=93ATTR_MAP
inputAttributeMapTableKeyCol=SOURCENAME
inputAttributeMapTableValCol=ATTRIBUTENAME
```

## Step 4: Create the BulkLoader Configuration File

You configure the BulkLoader utility by creating a configuration file for it that has the properties described in this section. You can name the file anything you want.

You set the properties in the file according to the following syntax:

```
property=value
```

**Note**

All property names and values in the configuration file are case-sensitive.

The following table describes properties in a BulkLoader configuration file:

| Property Name | Required/ Optional | Comments |
|---|---|---|
| `maxThreads` | Required | The maximum number of concurrent processing threads. This can be the number of database connections to the Content Server database server. Use as many threads as the number of CPUs on the database host. For a single CPU database host, set it to 2.<br>Example: `4` |
| `dataSliceSize` | Required | Number of items retrieved in one read request; this number will also be processed by a single processing thread.<br>Example: `2000` |
| `dataExtractionImplClass` | Required | User-specific Implementation class for data extraction API. Needs a constructor with (`String configFilename`) signature. The one mentioned here is a reference implementation class for backward compatibility. Data in flat tables.<br>Default value (“out of the box”): `com.openmarket.gatorbulk.objects.DataExtractImp` |
| `initId` | Required | Starting Content Server ID used the very first time BulkLoader operates; subsequently will use the value from `idSyncFile`.<br>Example: `800000000000` |
| `idSyncFile` | Required | Next available Content Server ID is saved in this file; updated during a BulkLoader session<br>Example:<br>`C:\FutureTense\BulkLoaderId.txt` |
| `idPoolSize` | Required | Each time BulkLoader needs to generate Content server IDs, it collects this many IDs and caches in memory. A good estimate is (number of assets * average number of attributes *2).<br>Example: `1000` |
| `commitFrequency` | Required | Number of flex asset groups to be part of a database transaction.<br>Example: `100` |

| Property Name | Required/ Optional | Comments |
|---|---|---|
| `outputJdbcDriver` | Required | The name of the JDBC driver class to access the Content Server database. The value here reflects the Oracle 9.0 driver.<br>Example:<br>`oracle.jdbc.driver.OracleDriver` |
| `outputJdbcURL` | Required | The JDBC URL. The following example value is a typical type2 oracle JDBC driver URL:<br>`Jdbc:oracle:oci8:@foo` |
| `outputJdbcUsername` | Required | Content Server database user name |
| `outputJdbcPassword` | Required | Content Server database user password |
| `inputTable` | Required | Name of the flat, input table from which new asset data is inserted |
| `inputAttributeMapTable` | Required | Name of the mapping table that lists the source table columns and the corresponding attribute names. |
| `inputAttributeMapTableKeyCol` | Required | The name of the column in the mapping table that lists the source table column names.<br>For example:<br>`inputAttributeMapTableKeyCol=SOURCENAME` |
| `inputAttributeMapTableValCol` | Required | The name of the column in the mapping table that lists the corresponding attribute names.<br>For example:<br>`inputAttributeMapTableValCol=ATTRIBUTENAME` |
| `inputTableDescriptionColumn` | Required | The name of the column in the source table that contains the descriptions of the flex assets.<br>For example:<br>`inputTableDescriptionColumn=SNSDESCRIPTION` |
| `inputTableGroupsColumn` | Required | Name of the column in the source table that contains the names of parents.<br>Each value can include several parents, separated by the `multivalueDelimeter` character, which is defined in the configuration file.<br>For example:<br>`inputTableGroupsColumn=SNSGROUP` |

| Property Name | Required/ Optional | Comments |
|---|---|---|
| `inputTableNameColumn` | Required | The name of the column in the source table that contains the name of the product (or advanced article or advanced image) for each row.<br>For example:<br>`inputTableNameColumn=SNSNAME` |
| `inputTableTemplateColumn` | Required | The name of the column in the source table that contains the flex definitions.<br>For example:<br>`inputTableTemplateColumn=SNSTEMPLATE` |
| `createdby` | Required | The user name that you want to be entered in the `createdby` field for your flex assets.<br>For example:<br>`createdby=editor` |
| `initId` | Required | The seed value for the first asset ID of the first flex asset that the BulkLoader utility imports. It starts with this value, incrementing for each asset ID that it creates.<br>For more information, see ""Setting the initID Parameter" on page 473. |
| `multivalueDelimeter` | Required | The delimiter that separates multiple attribute values. The default character is the semicolon (**;**).<br>For example:<br>`multivalueDelimiter=;` |
| `siteName` | Required | The name of the site. All products will behave as if they were created under this site.<br>For example:<br>`siteName=GE Lighting` |
| `status` | Required | The status code for all imported flex assets. You should set this to `PL` for imported.<br>For example:<br>`status=PL` |
| `tableProducts` | Required | Name of the flex asset type as defined in the Content server database. |

| Property Name | Required/ Optional | Comments |
|---|---|---|
| `inputTableUniqueIdColumn` | Required | Name of the column in the source table that serves as a unique identifier when importing a new flex asset. This will be used for any subsequent updates and void operations.<br>Leave this value empty, if you want the BulkLoader to generate unique identifiers for you |
| `targetName` | Required | Name of the publish target, as defined in Content Server |
| `renderTemplate` | Optional | Name of the template used for rendering flex assets - *deprecated* |
| `inputFeedbackTable` | Required | Name of the table that BulkLoader creates and uses for recording the processing feedback for every input item that was processed. Note that this table is created in the input data source. |
| `inputTableForUpdates` | Optional | Needed only if an update action is specified when running the BulkLoader utility. Otherwise, this can be an empty value. This is the name of the source table that contains attributes and parents that need updates. |
| `inputTableForUpdatesUniqueIdColumn` | Optional | Needed only if an update action is specified when running the BulkLoader utility. This is the name of the column in the source table that specifies a unique identifier for the flex asset. |
| `inputTableForUpdatesDeleteGroupsColumn` | Optional | Needed only if update action is specified and you have one or more flex assets that need one or more parents to be deleted.<br>This is the name of the column in the source table that specifies the list of parents to be deleted. |
| `inputTableForUpdatesAddGroupsColumn` | Optional | Needed only if an update action is specified and you have one or more flex assets that need one or more parents to be added. This is the name of the column in the source table that specifies a list of parents to be added. |
| `inputLimitRows` | Optional | Needed only for testing. Limits the number of input items processed for each action (insert, void, or update). |
| `updatedby` | Optional | The user name that you want to be entered in the `createdby` field for your flex assets. For example: `updateby=editor` |

| Property Name | Required/ Optional | Comments |
|---|---|---|
| `updatedstatus` | Optional | The status code for all updated flex assets. This must be set to `ED`.<br>For example: `updatestatus=ED` |

## Setting the initID Parameter

The `initID` parameter is the seed value that the BulkLoader starts at and increments from when creating a unique asset ID for each asset. You must choose a seed value number that allows the BulkLoader to create a contiguous block of ID numbers that cannot cause ID conflicts with existing (or future) asset ID numbers that are generated *by* Content Server.

Currently, Content Server starts at 1 trillion for the asset IDs that it creates. To be sure that you won't have conflicts, select a number low enough that when the BulkLoader utility is done, the highest ID number is under 900,000,000,000.

The BulkLoader creates one asset for each row/column value in the data source table. Each output table row requires its own unique asset ID.

Use these guidelines to determine the approximate number of asset IDs that are created by the BulkLoader utility:

- Five rows for each flex asset, plus
- Two rows per attribute for each flex asset

For example, if your data source table contains the following:

- 10,000 product assets
- 20 attributes per product (as determined by the product definition)
- 10 inherited attributes per product (as determined by the product parent definitions)

Then you need to allow for the following number of IDs:

(5 x 10,000) + (2 x 30 x 10,000) = 50,000 + 600,000 = 650,000 asset IDs

If your `initID` value is 800,000,000,000, then the BulkLoader creates ID numbers ranging from 800,000,000,000 to approximately 800,000,650,000.

## Example Configuration File

The following is an example of the BulkLoader configuration file that you could use with the GE Lighting sample site.

```
# New BulkLoader configuration for backward compatibility
#
# input datasource configuration
inputJdbcDriver=sun.jdbc.odbc.JdbcOdbcDriver
inputJdbcURL=jdbc:odbc:access-db-conn
inputJdbcUsername=
inputJdbcPassword=
#
# Source tables
#
inputTable=PRD_FLAT_50000
```

```
inputAttributeMapTable=PRD_FLAT_ATTRIBUTE_MAP
inputAttributeMapTableKeyCol=SOURCENAME
inputAttributeMapTableValCol=ATTRIBUTENAME
#
# input column names
#
inputTableTemplateColumn=CCTemplate
inputTableNameColumn=CCName
inputTableDescriptionColumn=CCDescription
inputTableGroupsColumn=CCGroups
#
# Content Server database
#
#  This database is always used for looking up Attributes,
#  Product Types and Product Group Types.
#  Data is imported into this database.
#
outputJdbcDriver=oracle.jdbc.driver.OracleDriver
outputJdbcURL=jdbc:oracle:oci8:@foo
outputJdbcUsername=csuser
outputJdbcPassword=csuser
#
# Data-specific settings
#
siteName=GE Lighting
targetName=Mirror Publish to burst37
initId=800000000000
createdby=user_designer
status=PL
renderTemplate=CLighting Detail
MAX_ATTRIBUTES=100
multivalueDelimiter=;
commitFrequency=50
#
# The following denotes the flex asset type that we are importing.
tableProducts=Products
#
# Additional information needed for BulkLoader
maxThreads=2
# dataSliceSize 0 means read all input data in one slice.
dataSliceSize=500
dataExtractionImplClass=com.openmarket.gatorbulk.objects.DataExtra
ctImpl
idSyncFile=C:\\FutureTense50\\bulk_uniqueid.dat
idPoolSize=50000
# For inserts
inputTableUniqueIdColumn=
inputFeedbackTable=bulk_feedback
# For updates
inputTableForUpdates=prod_flat_2_upd
inputTableForUpdatesUniqueIdColumn=input_id
inputTableForUpdatesDeleteGroupsColumn=CCGroups
```

```
inputTableForUpdatesAddGroupsColumn=
inputLimitRows=1000
##########################################################
```

## Step 5: Run the BulkLoader Utility

Before you begin, be sure that you have the appropriate JDBC drivers for **both** your **source** database and your **target** Content Server database.

Complete the following steps:

1. Put the configuration file on a system from which you have access to both the Content Server database on the management system, and to your source database.
2. Stop the application server on the management system.
3. Enter the following command, all on a single line, with paths that are appropriate for your installation:

   **For UNIX**

   ```
   java -ms16m -mx256m -cp <path to gatorbulk.jar>/gatorbulk.jar:
      <path to commons-logging.jar>/commons-logging.jar:
         <path to cs-core.jar>/cs-core.jar:
         <path to source jdbc driver>/<source_jdbc_driver>:
         <path to target jdbc driver>/<target_jdbc driver>: <path
      to commons-codec-1.3.jar>/commons-codec-1.3.jar:<path to
      commons-httpclient-3.0-rc2.jar>/commons-httpclient-3.0-
      rc2.jar:<path to commons-lang-2.1.jar>/commons-lang-2.1.jar
      com.openmarket.gatorbulk.objects.BulkLoader
      config=<bulkloader_configfile>
      action=<insert|void|update|all>   validate=<yes|no>
   ```

   **For Windows**

   ```
   java -ms16m -mx256m -cp <path to gatorbulk.jar>\gatorbulk.jar;
      <path to commons-logging.jar>\commons-logging.jar;
         <path to cs-core.jar>\cs-core.jar;
         <path to source jdbc driver>\<source_jdbc_driver>;
         <path to target jdbc driver>\<target jdbc driver>;<path
      to commons-codec-1.3.jar>/commons-codec-1.3.jar;<path to
      commons-httpclient-3.0-rc2.jar>/commons-httpclient-3.0-
      rc2.jar;<path to commons-lang-2.1.jar>/commons-lang-2.1.jar
      com.openmarket.gatorbulk.objects.BulkLoader
      config=<bulkloader_configfile> action=<insert|void|update>
      validate=<yes|no>
   ```

   Note that the action parameter specifies what BulkLoader needs to do: `insert`, `void`, or `update`. Setting the validate parameter to `yes` makes BulkLoader do extra validations during updates and voids. You may also need to increase the memory for the Java VM, depending on the size of your input data.
4. Examine the screen output to be sure that the BulkLoader utility was able to connect to the appropriate database.

### Step 6: Review Feedback Information

After the BulkLoader utility completes an operation, review the feedback information in the `bulk_feedback` table that is located in your input data source. That table contains information about all the input items that BulkLoader processed.

After reviewing that information, take any corrective actions that might be necessary. If you modify any of your input data, you should run BulkLoader again to verify that the errors were corrected.

### Step 7: Approve and Publish the Assets to the Delivery System

Use the BulkApprover utility to approve the assets that you just loaded. For instructions on how to use BulkApprover, see "Using BulkApprover" on page 488.

## Importing Flex Assets Using a Custom Extraction Mechanism

Sometimes users need alternative mechanisms to provide input asset data to BulkLoader. In such cases, the data may have to be gathered from multiple types of sources, such as XML documents, files, and legacy databases. To accomplish that, users can implement their own mechanism to provide data to BulkLoader, using the Java interface `com.openmarket.bulkloader.interfaces.IDataExtract`, which FatWire provides with Content Server.

A user can implement a Java object supporting `IDataExtract` and specify the Java object in the BulkLoader configuration file. BulkLoader will then invoke methods on this interface to initialize a read request, to repetitively read chunks of input data and then signal the end of the read request. This interface also has a method that provides import feedback from the BulkLoader utility, which can be used by the input provider to know the status of import and know any errors that may occur during import.

There are three Java interfaces that can help users with custom implementations of `IDataExtract`:

- `IDataExtract` – **Required** for any custom extraction.
- `IPopulateDataSlice` – Provides data to the BulkLoader utility. A container object supporting this interface is created by BulkLoader and passed into the client.
- `IFeedback` – Provides the status of each input item that has been processed by the BulkLoader. A feedback object that is created and populated by BulkLoader import thread is passed into the client.

These interfaces are described in the following sections.

**Note**

When you implement a custom extraction method, you use the same previously described procedures to run BulkLoader.

## IDataExtract Interface

This interface is **required** for any custom extraction.

The following is sample code that implements this interface.

com.openmarket.gatorbulk.interfaces.IDataExtract

```
package com.openmarket.gatorbulk.interfaces;
import java.util.Iterator;


/**
* To be implemented by input data provider.
* Interface for extracting data from an input source
* for BulkLoader.
* BulkLoader loads an object supporting this interface and invokes
* the GetNextInputDataSet() method on this interface repeatedly to
* fetch data in batches.
*/
public interface IDataExtract {

    public  final int HAS_DATA    = 100;
    public  final int NO_DATA = 101;

    public  final int SUCCESS = 0;
    public  final int ERROR = -1;

    public  final int INSERT_ASSETS = 1000;
    public  final int VOID_ASSETS = 1010;
    public  final int UPDATE_ASSETS = 1020;
    public  final int NONE_ASSETS = 1030;

    /**
    *Begin requesting input data; tells the client to
    *start the database query, get a cursor, etc.
    *@param requestType
    IDataExtract.INSERT_ASSETS,IDataExtract.VOID_ASSETS,
    IDataExtract.UPDATE_ASSETS
    *@param sliceOrNot  true/false
    * true - if data will be requested in batches
    * false - data will be requested all in one attempt
    *@param sliceSize  >0 number of rows to be
    *retrieved in one data set
    *@return none
    *@exception java.lang.Exception
    */
    public void InitRequestInputData(int requestType,
    boolean sliceOrNot, int sliceSize) throws Exception ;

    /**
    *Get a set/slice of input data records.
    *@param dataSlice object to be populated using the
```

```
    *methods from IPopulateDataSlice
    *@return IDataExtract.HAS_DATA when dataSlice has some data,
    *        IDataExtract.NO_DATA when there is no data,
    *        IDataExtract.ERROR when there is an error
    *@exception java.lang.Exception
    */
    public int  GetNextInputDataSet(IPopulateDataSlice dataSlice)
    throws Exception;

    /**
    * Signal the end of extracting data for given request type
    *@param requestType
    IDataExtract.INSERT_ASSETS,IDataExtract.VOID_ASSETS,
    IDataExtract.UPDATE_ASSETS
    *@return none
    *@exception java.lang.Exception
    */
    public void EndRequestInputData(int requestType)
    throws Exception;
    /**
    *Update the client as to what happened to input data
    *processing. Note that this method would be called by multiple
    *threads, with each thread passing its own IFeedback
    *handle. The implementor of this method should write
    *thread-safe code.
    *@param requestType
IDataExtract.InsertAsset,IDataExtract.VoidAsset,IDataExtract.Updat
eAsset
    *@param processingStatus - An object containing processing
    *status for all items in one dataset. The implementor of this
    *interface should invoke the IFeedback interface
    *methods on processingStatus to get status for individual
    *rows. This method will be invoked by multiple BulkLoader
    *threads, so make sure this method is implemented in a
    *thread-safe way.
    *@return none
    *@exception java.lang.Exception
    */
    void  UpdateStatus(int requestType, IFeedback
processingStatus) throws Exception;

}
```

## Implementation Notes for IDataExtract

The Java object implementing `IDataExtract` needs to have a constructor with a string parameter. BulkLoader will pass the name of its configuration file to the constructor when instantiating this object.

The method `UpdateStatus(..)` is invoked by multiple BulkLoader threads, so the implementation of this method should be thread-safe.

The following table lists and describes the configuration parameters for the BulkLoader utility when using custom data extraction method:

| Property Name | Required/ Optional | Comments |
|---|---|---|
| `maxThreads` | Required | The maximum number of concurrent processing threads. This can be the number of database connections to the Content Server database server. Use as many threads as the number of CPUs on the database host. For a single CPU database host, set it to 2.<br>Example: `4` |
| `dataSliceSize` | Required | Number of items retrieved in one read request; this number will also be processed by a single processing thread.<br>Example: `2000` |
| `dataExtractionImplClass` | Required | User-specific Implementation class for data extraction API. Needs a constructor with (String configFilename) signature. The one mentioned here is a reference implementation class for backward compatibility. Data in flat tables.<br>Default value (“out of the box”): `com.openmarket.gatorbulk.objects.DataExtractImpl` |
| `initId` | Required | Starting Content Server ID used the very first time BulkLoader operates; subsequently will use the value from `idSyncFile`.<br>Example: `800000000000` |
| `idSyncFile` | Required | Next available Content Server ID is saved in this file; updated during a BulkLoader session<br>Example:<br>`C:\FutureTense\BulkLoaderId.txt` |
| `idPoolSize` | Required | Each time BulkLoader needs to generate Content server IDs, it collects this many IDs and caches in memory. A good estimate is (number of assets * average number of attributes *2).<br>Example: `1000` |

| Property Name | Required/ Optional | Comments |
|---|---|---|
| `commitFrequency` | Required | Required.<br>Specifies when "`COMMIT`" statements will be inserted into the generated SQL file. A value of 0 means that "`COMMIT`" statements will be inserted every 50 lines (the default); any positive integer specifies the number of lines between each "`COMMIT`" statement.<br>For example:<br>`commitFrequency=5`<br>(A "`COMMIT`" statement will be inserted for every 5 lines of SQL code.) |
| `outputJdbcDriver` | Required | The name of the JDBC driver class to access the Content Server database. The value here reflects the Oracle 9.0 driver.<br>Example:<br>`oracle.jdbc.driver.OracleDriver` |
| `outputJdbcURL` | Required | The JDBC URL. The following example value is a typical type2 oracle JDBC driver URL:<br>`Jdbc:oracle:oci8:@foo` |
| `outputJdbcUsername` | Required | Content Server database user name |
| `outputJdbcPassword` | Required | Content Server database user password |

## IPopulateDataSlice

The following is sample code that implements this interface:

```
com.openmarket.gatorbulk.interfaces.IPopulateDataSlice

package com.openmarket.gatorbulk.interfaces;

import java.sql.Timestamp;


/**
*To be implemented by FatWire Corporation
*Interface to populate a dataSlice by the client.
*BulkLoader creates an object implementing this interface and then
*hands it over to the client, which uses this interface's methods
*to populate that object with input data records.
*/
public interface IPopulateDataSlice {

/**

*Creates a new input data object to hold all the data for a
*flex asset and makes it the current object. This method is
*invoked repetitively to populate this object with flex asset
*input data. Each invocation is to be followed by Set..()
*methods and AddAttribute..() methods to supply data for one
*flex asset.
*
*/
public void  AddNewRow();
    /**
    *Specify a unique identifier for flex asset input data
    @param id  user-specific unique identifier
    *@exception java.lang. Exception thrown if any unique-id
    *validation is enabled.
    */
public void SetAssetUniqueId(String id)  ;

    /**
    *Specify the name of the site with which the current flex
    *asset is created or to be created under.
    @param sitename  name of the site
    */
public void SetSiteName(String sitename)  ;
    /**
    *Set the asset type for the flex asset.
    @param flexAssetType asset type as defined in Content Server
system
    */
public void SetFlexAssetType(String flexAssetType)  ;
    /**
    *Specify the name of the parent for the current flex asset.
```

```
    *Use this method repeatedly to add a list of parent names.
    @param groupName  name of a parent that the current asset
    inherits some of its attributes from.
    */
public void AddParentGroup(String groupName)  ;
    /**
    *Specify the name of the parent to be deleted for the current
    *flex asset.
    *Use this method repeatedly to add a list of parent names.
    *@param groupName - name of a parent that the current asset
    *inherited some of its attributes from.
    */
public void AddParentGroupForDelete(String groupName);

    /**
    *Specify definition asset name for the current flex asset.
    @param definitionAssetName name of the flex definition asset
    */
public void SetDefinitionAssetName(String definitionAssetName)
;
    /**
    *Specify name of the flex asset.
    *@param name - name of the flex asset.Should be unique in
    *a flex asset family
    */
public void SetAssetName(String name)  ;
    /**
    *Specify description for the flex asset
    @param description  description
    */
public void SetAssetDescripiton(String description)  ;
    /**
    *Specify Content Server username with which this flex asset is
     being *processed
    @param username  Content Server username
    */
public void SetCreatedByUserName(String userName)  ;
    /**
    *Set Content Server status code for this asset
    *@param status
    */
public void SetAssetStatus(String status)  ;
    /**
    * Set template name
    *@param template  Content Server template name
    */
public void SetRenderTemplateName(String template)  ;
    /**
    *Specify startMenu for workflow participation
    *@param startMenuName  start menu name for this flex asset
    */
public void SetStartMenuName(String startMenuName)  ;
```

```
    /**
Content Server    *Specify publish approval target name
    *@param targetName  approval target name
    */
public void SetApprovalTargetName(String targetName)  ;
    /**
    *Add a name/value pair to specify a Content Server attribute
    of type  'text' for the current input object.
    *Call this method more than once, if this is a
    *multi-valued attribute.
    *@param attrName  attribute name as defined in the Content
     Server
    *database for the flex asset being processed
    *@param value  java.lang.String
    */
public void AddAttributeValueString(String attrName, String value)
;
    /**
    *Add a name/value pair to specify a Content Server attribute
of type
    *'date' for the current input object.
    *Call this method more than once, if this is a
    *multi-valued attribute.
    *@param attrName  attribute name as defined in the Content
    Server *database for the flex asset being processed
    *@param value  java.sql.Timestamp
    */
public void AddAttributeValueDate(String attrName, Timestamp
value) ;
    /**
    *Add a name/value pair to specify an attribute for the current
    *input object.
    *Call this method more than once, if this is a multi-valued
    *attribute
    *@param attrName  attribute name as defined in Content Server
    database *for the flex asset being processed
    *@param value  java.lang.Double
    */
public void AddAttributeValueDouble(String attrName, Double
value) ;
    /**
    *Add a name/value pair to specify a Content Server attribute
    of type *'money' for the current input object
    *Call this method more than once if this is a
    *multi-valued attribute
    *@param attrName  attribute name as defined in Content Server
database
    *for the flex asset being processed
    *@param value  java.lang.Float
    */
public void AddAttributeValueFloat(String attrName, Float value) ;
    /**
```

```
    *Add a name/value pair to specify a Content Server attribute
of type
    *'int' for the current input object.
    *Call this method more than once, if this is a
    *multi-valued attribute.
    *@param attrName  attribute name as defined in Content Server
    *database for the flex asset being processed
    *@param value  java.lang.Integer
    */
public void AddAttributeValueInteger(String attrName, Integer
value) ;
    /**
    *Add a name/value pair to specify any Content Server attribute
for the
    *current input object.
    *Use the datatype-specific methods above instead of this
    *method, as this one is for
    *supporting any other new types in future.
    *Call this method more than once, if this is a
    *multi-valued attribute
    *@param attrName  attribute name as defined in the Content
Server
    *database for the flex asset being processed.
    *@param value  java.lang.Object
    */
public void AddAttributeValueObject(String attrName, Object
value) ;
}
```

## IFeedback Interface

The following is sample code that implements this interface:

```
com.openmarket.gatorbulk.interfaces.IFeedback

package com.openmarket.gatorbulk.interfaces;

import java.util.Iterator;

/**
*To be implemented by FatWire Corporation
*Interface for the BulkLoader client to get the status of
*processing request to insert/void/update flex assets.
*/
public interface IFeedback {
    public final int ERROR=-1;
    public final int SUCCESS=0;
    public final int NOT_PROCESSED=1;
    /**
    *Get a list of keys from input data slice that has
    *been processed
    *@return java.util.Iterator
    */
public Iterator GetInputDataKeyValList();
    /**
    * Get Content Server asset ID for given input identifier
    *@param inputDataKeyVal key value of the unique identifier
    *in the input data record
    *@return Get the associated asset ID from the Content Server
system.
    *null if missing.
    */
public String GetContent ServerAssetId(String inputDataKeyVal);
    /**
    *Get the processing status for the input data record
    *identified by a key
    *@param inputDataKeyVal key value of the unique identifier
    *column in the input data record
    *@return ERROR - processed but failed, SUCCESS - processed
    *successfully, NOT_PROCESSED - unknown item or not part of
    *the processing dataset.
    */
public int  GetStatus(String inputDataKeyVal);
    /**
    *Get the associated error message for a given key,
    *unique identifier in input data
    *@param inputDataKeyVal  unique identifier for input data
    *@return error message, if GetStatus() returned ERROR
    *or NOT_PROCESSED
    */
public String GetErrorDescription(String inputDataKeyVal);
}
```

# Approving Flex Assets with the BulkApprover Utility

BulkApprover is a utility that quickly and easily approves large numbers of flex assets that you have loaded into the system via BulkLoader.

BulkApprover can perform the following tasks:

- Notify the approval system of all updates and deletions that were made during a previous BulkLoader session.
- Approve all newly loaded flex assets for one or more publishing targets.
- Mark all newly loaded flex assets as "published" for a given mirror destination, without actually publishing the assets. For example, to bypass a long mirror publishing session, you can instead copy selected assets from the content management database to a mirror destination on the delivery system and have BulkApprover mark the assets as "published" to the mirror destination.

**Note**

Only users with the `xceladmin` role can run BulkApprover.

## Configuring BulkApprover

Before running BulkApprover for the first time, you must create a configuration file for the utility. You can create a separate `BulkApprover.ini` file on a system that can access Content Server's database, or you can append the BulkApprover configuration information to one of BulkLoader's `.ini` files.

The following table lists required and optional configuration parameters:

| Parameter | Description |
|---|---|
| `bulkApprovalURL`<br>(Required) | The URL on the host server that has the data imported with BulkLoader.<br>The correct value is as follows:<br>`http://<myServer>/cs/ContentServer?pagename=OpenMarket/Xcelerate/Actions/BulkApproval`<br>where `<myServer>` is the name of the host server. |
| `adminUserName`<br>(Required) | The Content Server username of a user with the `xceladmin` role. |
| `adminUserPassword`<br>(Required) | The password for `adminUserName`. |

| Parameter | Description |
|---|---|
| `approvalTargetList`<br>(Required) | A list of destinations that the assets are to be approved for. For destination names, see the **Publish** option on the **Admin** tab, or the name column of the `pubtarget` table. Separate each destination with the delimiter that you specify in the `multiValueDelimiter` parameter. The syntax is:<br>*name1<multiValueDelimiter>name2 <multiValueDelimiter>name3* |
| `multiValueDelimiter` `(Required)` | A delimiter that you select. Use this delimiter to separate the approval targets that you specify in the `appovalTargetList` parameter. |
| `assetIdSqlFilter`<br>(Optional) | A statement that can be appended to a SQL `WHERE` clause in order to filter asset IDs.<br>For example:<br>`asset_id%20=0`<br>or<br>`asset_id%20!=0` |
| `debug`<br>(Optional) | Turns BulkApprover's debugging on and off.<br>A value of `true` turns debugging on. Leave this parameter blank for no debugging.<br>Debug messages are written to the file specified in the `output_file` parameter of the command line. |
| `assetschunksize`<br>(Optional) | Specifies the number of assets that are approved in a single transaction. For example, setting this property to `20` means that assets will be approved in groups of `20`.<br>Setting this property helps prevent session timeouts.<br>Default value: `25` |
| `outputJdbcDriver`<br>(Required) | The name of the JDBC driver class to access the Content Server database.<br>Example: `oracle.jdbc.driver.OracleDriver` |
| `outputJdbcURL`<br>(Required) | The JDBC URL. The following example value is a typical type 2 oracle JDBC driver URL:<br>`Jdbc:oracle:oci8:@foo` |
| `outputJdbcUsername`<br>(Required) | Content Server database user name. |
| `outputJdbcPassword`<br>(Required) | Content Server database user password. |

### Sample BulkApprover.ini File

The following sample shows the proper syntax of the BulkApprover configuration parameters:

```
bulkApprovalURL=http://MyServer/cs/
ContentServer?pagename=OpenMarket/Xcelerate/Actions/BulkApproval
adminUserName=admin
adminUserPassword=xceladmin
approvalTargetList=Dynamic;;;;;;testdest
multiValueDelimiter=;;;;;;
assetIdSqlFilter=
assetsChunkSize=3
debug=true
outputJdbcDriver=oracle.jdbc.driver.OracleDriver
outputJdbcURL=jdbc:oracle:thin:@19zln:1521:MyServer
#outputJdbcUsername=izod10
outputJdbcUsername=ftuser3
outputJdbcPassword=ftuser3
```

## Using BulkApprover

After you have configured the BulkApprover utility, you can use it to approve assets that were imported into the database via the BulkLoader utility.

BulkApprover runs from the command line. Parameters are described in the following table and included in the example below:

| Command-Line Parameter | Description |
|---|---|
| `config` | The name of the file that stores your BulkApprover configuration information; for example, `BulkApprover.ini`. |
| `action` | The action or actions that you want BulkApprover to perform. You must set this parameter to `notify` or `approve`. Add other actions, as necessary. To have BulkApprover perform multiple actions, name the actions in a comma-separated list.<br>**Valid values**:<br>• `notify` – Notifies the approval system about all updates and voids processed during a previous BulkLoader session.<br>• `approve` – Instructs BulkApprover to approve all of the assets that it processes for the given publishing destination(s).<br>• `mark_publish` – Marks all of the assets that it processes as "published" to a given mirror destination, without actually publishing the assets. Specify the publishing targets in the `approvalTargetList` parameter in the BulkApprover configuration file (see "Configuring BulkApprover" on page 486). If you do not want the assets marked as published, do not include this parameter. |
| `output_file` | The name of the log file that contains all output from the server; for example, `bulkapprover.txt`. |

To run BulkApprover, set paths as shown in the following example:

```
java -ms16m -mx64m -cp Path_to_gatorbulk.jar;
  Path_to_commons-logging.jar;Path_to_cs-core.jar;
  Path_to_cs.jar;Path_to_commons-codec-1.3.jar;
  Path_to_commons-httpclient-3.0-rc2.jar;
  Path_to_commons-lang-2.1.jar com.openmarket.gatorbulk.
   objects.BulkApprover
  config=bulkapprove.ini
  action=notify,approve,mark_publish
  output_file=bulkapprover.txt
```

Part 4

# Site Development

This part describes how to program your online site to deliver the data (assets) that you have designed.

It contains the following chapters:


- Chapter 23, “Creating Template, CSElement, and SiteEntry Assets”
- Chapter 24, “Creating Templates to Support Graphical Page Design”
- Chapter 25, “Creating Collection, Query, Stylesheet, and Page Assets”
- Chapter 26, “Best Practices for Creating Site Preview Assets and Templates”
- Chapter 27, “Coding Elements for Templates and CSElements”
- Chapter 28, “Template Element Examples for Basic Assets”
- Chapter 29, “Configuring Sites for Multilingual Support”
- Chapter 30, “Setting Up Flash Content Management”
- Chapter 31, “User Management on the Delivery System”
- Chapter 32, “The HelloAssetWorld Sample Site”
- Chapter 33, “The Burlington Financial Sample Site”

Chapter 23

# Creating Template, CSElement, and SiteEntry Assets

The CSElement, Template, and SiteEntry asset types provide the pagelets and elements that build your online sites. They are asset representations of page names and elements, the components that Content Server uses to generate pages.

When you create a CSElement asset, you code an element. When you create a SiteEntry asset, you name a page. When you create a template, you do both: you code an element and you name a page.

This chapter describes these three asset types and provides information about how to create them. Additional information about coding templates and CSElements is included in Chapter 27, "Coding Elements for Templates and CSElements."

This chapter contains the following sections:


- Overview
- Pages, Pagelets, and Elements
- CSElement, Template, and SiteEntry Assets
- Creating Template Assets
- Creating CSElement Assets
- Creating SiteEntry Assets
- Managing Template, CSElement, and SiteEntry Assets
- Using Content Server Explorer to Create and Edit Element Logic

# Overview

This chapter contains procedures for creating Template, CSElement, and SiteEntry assets.

- Template assets are classified as typed or typeless depending on whether they apply to a single asset type or no asset type.
- If you are using SiteLauncher (to replicate sites or share Template and CSElement assets), Content Server requires element logic to *indirectly* refer to assets, asset types, attribute names, and template names. To this end, the CS interface introduces the **Map** screen (for example, page 517); the API introduces the `render:lookup` tag.

  Using the **Map** screen, you assign an alias to each value. You can then hardcode the aliases in the element logic and use the `render:lookup` tag to retrieve the actual values from the aliases at runtime.
- The **Cache Rules** field has been simplified to reduce errors. Template developers can choose "cached," "uncached," or "advanced." Selecting **Advanced** allows developers to set caching rules individually for Content Server and Satellite Server.
- A new tag, `calltemplate`, was introduced to invoke templates in a way that simplifies the template writing process.
- The **PageCriteria** field has been renamed to **Cache Criteria**. It accepts the following reserved parameters: `c, cid, context, p, rendermode, site, sitepfx, ft_ss`, and custom-defined parameters.

  Cache criteria values are stored in the `pagecriteria` column of the `SiteCatalog` table (in previous versions they were stored in the `resargs` columns of the `SiteCatalog` table).

  The **Cache Criteria** field is also used to hold variables that enable the "Extra Parameters" section in the FCKEditor and make them available to users, in the **Include asset link** and **Add asset link** pop-up windows. The "Extra Parameters" section provides a way of passing custom parameters (such as image dimensions) to the template. See page 514 for more information about extra parameters. See also the Content Server Advanced and Dash user guides.
- Forms for creating Template and CSElement assets have been subdivided by tabs; fields are organized by function on the tabs.

# Pages, Pagelets, and Elements

In the Content Server context, an online page is the composition of several components into a viewable, final output. Creating that output is called **rendering**. (Making either that output or the content that is to be rendered available to the visitors on your public site is called publishing.)

Content Server renders pages by executing the code associated with page names. The name of a page is passed to Content Server from a browser and Content Server invokes the code associated with that page name. The code is actually a named file, a separate chunk of code called an element.

The code in your elements identifies and then loads assets to display in those pages or pagelets, and passes other page names and element names to Content Server. When Content Server invokes an element, all of the code in the element is executed. If there are calls to other elements, those elements are invoked in turn. Then the results—the images, articles, linksets, and so on, including any HTML tags—are rendered into HTML code (or some other output format if your system is configured to do so).

Template, CSElement, and SiteEntry assets represent elements and pagelets as follows:

- A CSElement asset is an element.
- A SiteEntry asset is the name of a page or a pagelet.
- A Template asset is both an element and a page or pagelet that renders an asset.

## Elements, Pagelets, and Caching

Pages and pagelets are cacheable. They have cache criteria set for them that determines whether they are cached and, if so, for how long.

Elements do not have cache criteria. When your code calls an element directly by name, without going through a page name, the output is displayed in the page that called the element's name and that output is cached as a part of that page.

If you want to cache the output from an element separately from the output of the page that called it, you must provide a page name for it and call it by its page name. The code in a Template asset has a page name by default. To provide a page name for a CSElement asset, you create a SiteEntry asset and select the CSElement asset for it.

## Calling Pages and Elements

To see a Content Server page, you provide a URL that includes the name of the page. A Content Server URL looks like this:

- For WebLogic, WebSphere, and Sun ONE Application Server:

  `http://`*`host`*`:`*`port`*`/`*`servlet_context_path`*`/ContentServer?pagename=`*`name_of_page`*

  where

  *`host`* is the name of the server that is hosting the Content Server system,

  *`port`* is the port number of the web server,

  *`servlet_context_path`* is the path that the application server gives to the Content Server web application, and

  *`name_of_page`* is the page name.

This syntax passes the name of a page to the ContentServer servlet, which then renders the page.

For example, to see the home page of the Burlington Financial sample site, you enter:

```
http://127.0.0.1:7001/servlet/ContentServer?pagename=
  BurlingtonFinancial/Page/Home
```

When you code your elements, you use tags that programmatically call the pagelets and elements that you want to display in your site. These tags pass the names of pages and elements to the ContentServer servlet just as a URL entered in a browser passes a page name to the ContentServer servlet.

To call a page name, use the `render:satellitepage` (`RENDER.SATELLITEPAGE`) tag. For example:

```
<render:satellitepage pagename="BurlingtonFinancial/Page/Home"/>
```

To call an element directly by name, use the `render:callelement` (`RENDER.CALLELEMENT`) tag. For example:

```
<render:callelement elementname="BurlingtonFinancial/Common/
  TextOnlyLink"/>
```

To call a template by name, use the `render:calltemplate` tag. For example:

```
 <render:calltemplate
        site='<%=ics.GetVar("site")%>'
        slotname="Head"
        tid='<%=ics.GetVar("tid")%>'
        c='<%=ics.GetVar("c")%>'
        cid='<%=ics.GetVar("cid")%>'
        tname='<%=ics.GetVar("HeadVar")%>'>
    <render:argument name="p" value='<%=ics.GetVar("p")%>' />
</render:calltemplate>
```

**Note**

When you use CS-Explorer to examine `SiteCatalog` and `ElementCatalog` entries, they are presented as folders and subfolders that visually organize the pages and pagelets.

However, these entries are simply rows in a database table—there is no actual hierarchy. Therefore your code must always call a page entry or an element entry by its entire name. You cannot use a relative path.

How does your code call template, CSElement, and SiteEntry assets? As follows:

- Because a SiteEntry is a pagelet, you use the `render:satellitepage` tag to call SiteEntry assets from within your element code.
- Because a CSElement is an element, you use the `render:callelement` tag to call CSElement assets from within your element code.
- Because a template is both an element and a page name, you can use either of the above, although typically the `render:calltemplate` tag is designed to be used for templates. It encapsulates the functionality of `render:satellitepage` and `render:calleelement` as well as other features, such as parameter validation.

## Page vs. Pagelet

For the sake of clarity, the following table lists the various terms that include the word "page" and defines them in the context of their usage in the documentation for Content Server, the CS modules, and the products:

| Term | Definition |
|---|---|
| pagelet | The results of an HTTP request displayed in a browser as one piece of a rendered page. It has an associated element file.<br>A pagelet can be cached in the Content Server and Satellite Server page caches. |
| page | The results of an HTTP request displayed in a browser window. A page is created by compiling several parts of pages (pagelets) into one final, displayed or rendered page. It has an associated element file.<br>A page can be cached in the Content Server and Satellite Server page caches. |
| page name | The complete name of a page or pagelet. For example: `BurlingtonFinancial/Article/Full`. |
| page asset | Page assets do **not** represent page names. They represent logical containers for content. These containers can be arranged into a tree structure for navigation of site content.<br>You create page assets and then place them in position in the "Site Plan" tree which is visible on the **Site** tab in the tree in the left pane of the Content Server interface. You associate other content and site design assets with them and then you publish them. |

# CSElement, Template, and SiteEntry Assets

As mentioned, CSElement assets are elements, SiteEntry assets are page names, and Template assets include both.

Because page names and elements are assets, you can manage your code and page names in the same way you manage your content: you can use workflow, revision tracking, approval, and preview, as well as the Mirror to Server publishing method to move your code and page names to the management and delivery systems.

**Caution**

**Revision tracking.** Never use the revision tracking feature in the CS-Explorer tool to enable revision tracking directly on the `SiteCatalog` or `ElementCatalog` tables.

**Mirror to Server.** If templates or CSElements refer to elements that are not associated with a template or CSElement asset, these elements are not automatically mirrored to the publishing destination. You must move them manually with the CatalogMover utility. For this reason, we do not recommend using elements that are not wrapped by CSElements.

When you create a CSElement or Template asset, you create an element. You can code the element by using the asset form (**Template** or **CSElement**, depending on which type of asset you are creating).

**Note**

Elements for Template assets and CSElements can be coded in CS-Explorer. However, the procedure is not recommended for reasons dealing mostly with compositional dependencies and updates to the cache. Developers who prefer to use CS-Explorer must follow the steps in "Using Content Server Explorer to Create and Edit Element Logic" on page 540 in order to ensure the validity of the Template or CSElement assets.

## Template Assets

Templates render other assets into pages and pagelets. This in turn creates the look and feel of your online site. You create a standard set of templates for each asset type—except CSElement and SiteEntry assets—so that all assets of the same type are formatted in the same way.

This process allows content providers to preview their content by selecting formatting code for the content, but not requiring them to code themselves or allowing them to change your standard, approved code.

When you save a Template asset, Content Server does the following:

- Creates a row in the `Template` table for the asset.
- Creates an element entry in the `ElementCatalog` table. The name of the entry uses the following convention:

  *`AssetTypeName/TemplateName`*

  where:

  - *`AssetTypeName`* is the asset type formatted by the Template asset and element.
  - *`TemplateName`* is the name of the template.
- Creates a page entry in the `SiteCatalog` table. The name of the page entry uses the following convention:

  *`SiteName/AssetTypeName/TemplateName`*

  where:

  - *`SiteName`* is the name of the site that the template belongs to, which is the site that you were working in when you created the template. CS-Direct obtains this name from the `Publication` table. (In previous versions of the product, sites were called "publications.")
  - *`AssetTypeName`* is the asset type formatted by the Template asset and element
  - *`TemplateName`* is the name of the template.

**Note**

- Do not change the name of the page entry that CS-Direct creates.
- For backwards compatibility, CS 7.0.3 (and above) supports the page entry naming convention used in CS 6.1.

  In CS 6.1, page entries are named:
  *`SiteName/RootElementName`*

  In CS 6.3 and above page entries are named:
  *`SiteName/AssetTypeName/TemplateName`*

  In CS 7.0.3, if the root element name is prefixed with the site name (for example, if you upgraded from CS 6.1 to CS 7.0.3 and kept all your assets), CS will automatically remove the site name from the root element name when generating the page entry. This way, page entries will be named correctly, regardless of how the root element is named.

- Creates new rows in other tables that support the operation of the Template asset. The tables start with the name: `Template_`
- Creates a new row in the `AssetPublication` table to associate your template with your site.

## CSElement Assets

You use CSElement assets for the following kinds of things:

- Code that is not for rendering an asset and that you want to reuse in more than one place and/or call from more than one type of template. For example, you have six templates that use the same top banner so you create a CSElement asset for the code in the banner and call that element from each template. This way, if you decide to change the way the banner works, you only have to change it in one place.
- Recommendations for Engage. If you create a dynamic list recommendation, you must create a CSElement asset to build the dynamic list. For more information, see Chapter 40, “Recommendation Assets.” These assets do not render content, but exist for logic processing.

When you save a CSElement, Content Server does the following:

- Creates a row in the `CSElement` table for the asset.
- If you have coded the element in the “CSElement” form, creates an element entry in the `ElementCatalog` table. The name of the entry is the name that you entered into the **ElementCatalog Entry Name** field in the form.
- Creates a new row in the `AssetPublication` table to associate your template with your site.
- Creates a new row in the `AssetPublication` table to associate your template with your site.

## SiteEntry Assets

You use SiteEntry assets for the following kinds of things:

- If you are using the CS-Designer tool, you use SiteEntry assets to represent code snippets. In that interface, when you drag and drop a code snippet into a page, you are dropping in a Content Server call to a page entry through a `render:satellitepage` tag.
- When the code in a CSElement asset is rendered, the code is displayed in the page that called it, and is cached as part of that page (if that page is cached, that is). If you want the output from a CSElement to be cached as a separate pagelet and have its own cache criteria set for it (timeout value, page criteria values, and so on), your code must invoke that element through a page name. In such a case, you create a SiteEntry asset to accompany your CSElement asset.

When you create and save a SiteEntry asset, you associate a CSElement asset with it. The element in that CSElement asset becomes the root element for the SiteEntry's page entry.

When you save a SiteEntry asset, Content Server does the following:

- Creates a row in the `SiteEntry` table for the asset.
- Creates a page entry in the `SiteCatalog` table. The root element of the page entry is the element from the CSElement asset that you specified.
- Tracks an approval dependency between the SiteEntry asset and the CSElement asset. Both the SiteEntry asset and its CSElement asset must be approved before the SiteEntry asset can be published.

**Note**

Compositional dependencies are also tracked. The SiteEntry defines the page criteria and the default arguments that contain the dependency information. The CSElement records the `id` of the SiteEntry and CSElement assets into the rendering engine using `render:logdep` tags that are added to the CSElement code stub.

## What About Non-Asset Elements?

If you code customizations for the Content Server interface on the management system, you create elements that are not assets because you do not want them to be published to your delivery system.

For example, when you create workflow elements that implement actions or conditions, you do not create them as CSElement assets. Rather, you use the CS-Explorer tool to manually create an entry in the `ElementCatalog` table.

Remember that if you create workflow or other custom elements on your delivery system, you must use the CatalogMover utility to copy those elements to the `ElementCatalog` on your management system.

**Note**

You can write code to invoke the mirror engine to mirror your elements. The topic is advanced and beyond the scope of this guide. For code samples, visit our web site: `http://developernet.fatwire.com`

# Creating Template Assets

Templates render assets. When you create a Template asset, you create it as either

- a typed template, for rendering assets of a specific type, or
- a typeless template, which applies to assets of any type. A typeless template is generally used to specify the layout of a page in which assets can then be rendered by the typed templates.

**Note**

The only field that makes a template typed or typeless is the **For Asset Type** field (page 507). The purpose of distinguishing templates as typed or typeless is to help developers manage the construction of pages and easily keep track of which templates are responsible for page layout and which for asset rendering.

Before creating a Template asset, complete the "Prerequisites" section to determine how you will set template properties (such as the template name) and how you will code the template's element logic. You will then complete the following steps, using the Content Server interface:

- Step 1: Open the 'Template' Form
- Step 2: Name and Describe the Template Asset
- Step 3: Configure the Template's Element—To specify its usage, file type, and logic
- Step 4: Configure SiteEntry—To specify page and pagelet caching parameters
- Step 5: Configure the Map—If you wish to support template sharing and site replication
- Step 6: (Optional) Create a Thumbnail—To graphically represent the template in its **Inspect** form
- Step 7: Inspect the Template

Information that you enter into the **Template** form will be written to database tables when the template is saved.

**Note**

Do not create Template assets directly in the database tables. Doing so will require you to write to several tables and can result in incorrect tracking of dependencies. Instead, use the **Template** form and the procedures in this section to create Template assets. For help with coding the template's element logic (in typed templates), see Chapter 27, "Coding Elements for Templates and CSElements."

## Prerequisites

Before you begin creating a Template asset, determine the following:

- `TemplateName` (a name for your Template asset; the value of the **Name** field in the **Name** screen, page 507).
- Whether the Template asset is to be typed or typeless.

- Whether the Template asset will be shared and whether the site you are working in will be replicated. These considerations determine how you will code the template's element logic.
- Whether to code the Template's element logic in CS-Explorer instead of the **Template** form. (Coding in CS-Explorer, although practiced, is not recommended for the reasons outlined in "Using Content Server Explorer to Create and Edit Element Logic" on page 540.)

## Naming a Template Asset

- Once a Template asset is saved, its name cannot be changed.
- Content Server appends the template name to *SiteName* (also to *AssetTypeName* for typed templates). The template name should make sense in relation to *SiteName* and *AssetTypeName*. (Content Server's naming conventions must not be overridden; names created by CS-Direct must not be changed.) Table 13 lists the conventions that use *TemplateName*.

  *AssetTypeName* is used only for typed templates and represents the value of the template's **For Asset Type** field in the **Name** screen (page 507). *SiteName* is the name of the site to which the template belongs (the site that you are working in as you are creating the template). CS-Direct obtains the *SiteName* from the `Publication` table. (In previous versions of the product, sites were called "publications.")

**Table 13:** Naming Conventions Using *TemplateName*

| Template | Naming Conventions | Description |
|---|---|---|
| Typed | *AssetTypeName/TemplateName* | Name of the root element for a typed template.<br>This value is written to the **Rootelement** field in step 4 on page 512. **This value must not be changed.** If the default value is changed, some tags that expect the default value, such as the `render:calltemplate` tag with the `style` attribute set to `element`, will fail.<br>**Note:** The naming convention requires root element names to be unique. You must not have multiple Template assets pointing to the same root element. You can, however, have two SiteEntry assets point to the same element (for example, if you wish to specify different default arguments, or different cache criteria depending on the calling scenario). |
| | *AssetTypeName/ TemplateName*.xml_or_jsp_or_html | Path to the element file of a typed template.<br>This value is written to the **ElementStorage Path/Filename** when the file type is selected in step 3 on page 511. |
| | *SiteName/AssetTypeName/ TemplateName* | Name of the page that will be rendered if the template is typed.<br>This value is written to the **SiteCatalog Pagename** field, in "Step 4: Configure SiteEntry" on page 513. |

**Table 13:** Naming Conventions Using *TemplateName*

| Template | Naming Conventions | Description |
|---|---|---|
| Typeless | */TemplateName* | Name of the root element for a typeless template.<br>This value is written to the **Rootelement** field in step 4 on page 512. **This value must not be changed.** If the default value is changed, some tags that expect the default value, such as the `render:calltemplate` tag with the `style` attribute set to `element`, will fail.<br>**Note:** The *AssetTypeName* is omitted, as the template applies to any asset type. The forward slash is kept to identify the template as typeless. See also the note in the first row of this table. |
| | `Typeless/`*TemplateName*`.xml_ or_jsp_or_html` | Path to the element file of a typeless template.<br>This value is written to the **ElementStorage Path/Filename** when the file type is selected in step 3 on page 511. |
| | *SiteName/TemplateName* | Name of the page that will be rendered if the template is typeless.<br>This value is written to the **SiteCatalog Pagename** field, in "Step 4: Configure SiteEntry" on page 513.<br>**Note:** The *AssetTypeName/* is omitted, as the template applies to any asset type. |

## Designating a Template as Typed or Typeless

Before creating a Template asset, determine whether it is to be typed or typeless. Once the template is saved, its status as typed or typeless cannot be changed.

## Template Sharing and Site Replication

Before creating a template, decide how the template and the site you are working in will be used. Your decision determines how you will code the template's element logic (in "Step 3: Configure the Template's Element" on page 508).

If you wish to share your Template asset or make the current site replicable**,** ensure that the template's element logic does not directly refer to assets, asset types, attribute names, or template names. Instead, you must refer to them indirectly. Use the **Map** screen ("Step 5: Configure the Map" on page 517) to assign an alias ("key") to each value, then hard code the aliases in your template. Use the `render:lookup` tag to retrieve the actual values from the aliases at runtime.

During its execution, the `render:lookup` tag refers to the map to look up the keys and returns the asset-specific information for use in the element logic. This dynamic lookup allows the Template asset (but not the element logic alone) to refer directly to asset data while enabling safe replication and template sharing.

For example, assume a template is named `FSIILayout`, and the site containing this template has a site prefix of `FSII`. If the site is replicated such that the new site's prefix is `New`, and `the FSIILayout` template is copied, then the copy of the template is named `NewLayout`. Referring to the `NewLayout` template by its hard-coded name (`FSIILayout`) would result in a failure when the template is executed. Instead, the template name is looked up:

```
<%-- Look up the name of the layout template --%>
<render:lookup
        site='<%=ics.GetVar("site")%>'
        varname="LayoutVar"
        key="Layout"
        tid='<%=ics.GetVar("tid")%>'/>

<%-- Look up the name of the wrapper page's site entry.
     Note we want the asset name only, so we must specify
     the match filter. --%>
<render:lookup
        site='<%=ics.GetVar("site")%>'
        varname="WrapperVar"
        key="Wrapper"
        tid='<%=ics.GetVar("tid")%>'
     match=":x"/>
```

To code the element logic, you must have a clear understanding of its design and the map it will refer to. You will need to determine:

- Which keys to create and which name to assign to each key.
- The type of asset information to be looked up:
  - Template Name
  - Asset Type
  - Asset (Type:Name)
  - Asset (Type:ID)
- A value for each key.
- The site to which the map applies.

Additional information about usage of the `render:lookup` tag is given in the *Developer's Tag Reference* and in the *FirstSiteII* guide.

## Procedures for Creating Template Assets

This section shows you how to create a Template asset, using the Content Server interface.

**Note**

Before starting the procedures in this section, read "Prerequisites" on page 501 for information about creating Template assets.

### Step 1: Open the 'Template' Form

1. Log in to Content Server's Advanced interface as an administrator.

2. Select the site in which you want to work.
3. In the button bar, click **New**.
4. In the list of asset types, select **New Template**.

> **Note**
>
> For the **New Template** option to be displayed, the Template asset type must be enabled for your site and a start menu item must be created for it.

5. The **Template** form appears. Continue with "Step 2: Name and Describe the Template Asset."

> **Note**
>
> If you see a **Choose Assignees** screen instead of the **Template** form, it means that the Template asset you will be creating is associated with a workflow. Select a name (or names) from the "Users" column and click **Set Assignees**. Continue with "Step 2: Name and Describe the Template Asset."

## Step 2: Name and Describe the Template Asset

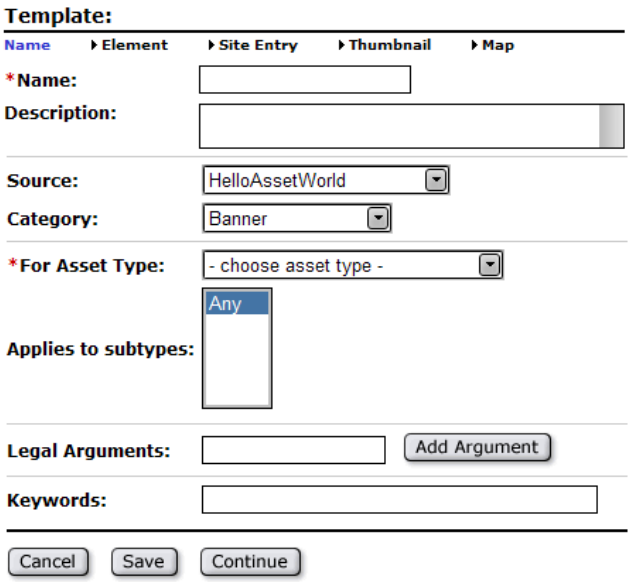
The **Name** screen is used to identify the template as typed or typeless, assign the template to a category, specify arguments that may be passed to the template, and name keywords by which the template can be located in search routines.

When the Template asset is saved, field values that you specify in the **Name** screen (with the exception of legal arguments), are written to the `Template` table, as indicated in the procedures below.

> **Note**
>
> At any time in the process of creating a template, you can save the template. Content Server will display the template's **Inspect** form. To return to the **Template** form, click the **Edit** link.

### To name and describe the Template asset

In the **Name** screen, fill in the fields as explained in the steps below:




1. (Required). In the **Name** field, type a descriptive template name that is unique for the template and for the type of asset(s) that the template renders. It is best to choose a name that reflects the function or purpose of the template.

   **Valid entries:**

   - Up to 64 alphanumeric characters (the first character must be a letter)
   - Underscores (_)
   - Hyphens (-)
   - Spaces (these will be converted to underscores when used in the `SiteCatalog` pagename for the template)

   **Note**

   Make sure you have chosen a name for your Template asset using the guidelines in the section "Prerequisites" on page 501.

2. In the **Description** field, type a brief description of the template. You can use up to 128 characters.
3. In the **Source** field, select an option from the drop-down list if your template is derived from a source that you wish to note.
4. In the **Category** field, select an option from the drop-down list if you wish to place the Template asset into a category.
5. (Required). In the **For Asset Type** field, identify your template as typed or typeless:

   - If you are creating a typeless template (for example to dispatch to typed templates), select **Can apply to various asset types** and skip to step 7.

- If you are creating a typed template (which renders assets of a certain type), select an asset type. For example, if you are creating a template to render article assets, select **Article** from the drop-down list.

6. (Required for typed templates). In the **Applies to Subtypes** field, select the appropriate subtypes from the drop-down menu.

> **Note**
>
> A typed template should be used only for specific subtypes of the asset type that you selected in the preceding field (**For Asset Type**).

7. In the **Legal Arguments** field:
   a. Enter an argument that may be passed to the template and click **Add Argument**.
   b. In the fields that are displayed:
      - Specify whether the argument is optional or required.
      - Provide a description of the argument (to help you know the purpose of the argument you are creating).
      - Specify legal values (including descriptions) for the argument.

   (You can specify as many arguments and legal values as you require by clicking the **Add Arguments** and **Add Legal Value** buttons.)
8. In the **Keywords** field, enter keywords that you and others can use as search criteria in the **Advanced Search** form when you search for this template in the future. For information about searching for assets, see the *Content Server Advanced Interface User's Guide*.
9. Click **Continue** to open the next screen, **Element**.

> **Note**
>
> If you chose to save the Template asset, you will notice that Content Server adds two fields:
>
> - The **Status** field, which is pre-populated with the editorial status of the Template asset ("created," "edited," and so on). This field identifies the latest operation that was performed on the Template asset, regardless of whether the Template asset is associated with a workflow.
> - The **ID** field, which is pre-populated with a unique number that Content Server generates and assigns to the Template asset as its ID. (The **ID** field corresponds to the `tid` variable.)

## Step 3: Configure the Template's Element

The **Element** screen (page 510) is used to create the template's element—define the element file type (XML, JSP, or HTML), provide the element logic, and name the element. For example:

- The **Create Template Element** field offers a choice of XML, JSP, or HTML file types for the element logic, and is used to seed the **Element Logic** field with standard stub code (which you need to include in any element that you create).

When you use the **Create Template Element** field to create, for example, a .jsp file, CS-Direct adds JSP `taglib` statements and the `RENDER.LOGDEP` tag to the **Element Logic** field by default so that Content Server can log the compositional dependency between this Template asset and pages that are rendered from this element. For other file types, CS-Direct adds code specific to the file type. You will add your own code to the **Element Logic** field.

For information about dependencies, see "About Dependencies" on page 580. For help with coding the element logic, see Chapter 27, "Coding Elements for Templates and CSElements."

- The **Element Storage Path/Filename** field names the file that holds the element logic and specifies the path to the file.

When the Template asset is saved, field values in the **Element** screen are written to a row (representing the element) in the `ElementCatalog` table, as indicated in the procedures below.

### Selecting an Existing Element

In the steps that follow, we assume you are creating a new element for the Template asset. If, however, you are migrating assets from an earlier Content Server release and wish to reuse an existing element, you need to identify the element correctly so that CS-Direct can find it and associate it with this Template asset.

**To select an existing element**

1. (Optional). In the **ElementCatalog Description** field, type a description of the element.
2. In the **Element Storage Path/Filename** field, enter a value according to the naming convention in Table 13, on page 503.
3. In the **Element parameters** field, specify the variables or arguments that can be passed to the element. For more information, see step 7 on page 512.
4. Save and re-open the Template asset.

   CS-Direct checks for the existence of the named element:

   - If the element has been correctly named, CS-Direct recognizes the element and displays its code in the **Element Logic** field.
   - If the named element does not exist (or is incorrectly named), CS-Direct does nothing. When you inspect or edit the Template asset, CS-Direct displays a message stating that there is no root element in the form. As soon as you code the element and give it the correct name, CS-Direct detects it and associates it with the template.

### To configure a new element

In the **Element** screen, fill in the fields as explained in this section.

**Template: MyTemplate**

Name | Element | Site Entry | Thumbnail | Map

**Usage:** Element is used within an HTML page.

**Called Templates:**
- choose templates this template calls -
FSIIAddToCart
FSIIBottomNav
FSIICartDetailView
FSIICartSideNavView

**Create Template Element?** XML JSP HTML

**Rootelement:** /MyTemplate

**ElementCatalog Description:**

**Element Storage Path/Filename:**

**Element Logic:**

**Element Parameters:**

**Additional Element Parameters:**

Cancel Save Continue

1. In the **Usage** field, specify the intended usage of this template, using the table below as a guideline.

| Usage Option | Description |
|---|---|
| Usage unspecified | Specifies a template that generates HTML. It is unknown whether the template is a "Body" template (see row 2 of this table) or a "url" template (see row 3 of this table). |
| Element is used within an HTML page | Specifies a template that is used inside the `<BODY>...</BODY>` tag of an HTML page. This option characterizes the template as a "Body" template. |
| Element defines a whole HTML page and can be called externally | Specifies a template that generates a complete HTML page and can be used in a url. This option characterizes the template as a "url" template. |
| Element is streamed as raw data | Specifies a template that generates raw binary data of an unknown type that is not HTML. |

2. In the **Called Templates** field, select the template(s) that this template will call (if they exist).
3. In the **Create Template Element** field, do one of the following:
   - To create an .xml file, click **XML**. The code that is pasted in comes from the `OpenMarket\Xcelerate\AssetType\Template\modelXML.xml` element and can be modified to use custom default logic.
   - To create a .jsp file, click **JSP**. The code that is pasted in comes from the `OpenMarket\Xcelerate\AssetType\Template\modelJSP.xml` element and can be modified to use custom default logic.
   - To create an .html file, click **HTML**. The code that is pasted in comes from the `OpenMarket\Xcelerate\AssetType\Template\modelHTML.xml` element and can be modified to use custom default logic.

   CS-Direct populates the following fields:
   - **Element Logic** field with a header and other auto-generated code.

     For example, if you clicked the **JSP** button, CS-Direct enters a tag library directive for each of the CS-Direct JSP tag libraries. Content Server also sets a `RENDER.LOGDEP` (`render:logdep`) tag to mark a compositional dependency between the Template asset and any page or pagelet rendered with the template.
   - **Element Storage Path/Filename** field. **Do not change the value of this field.**

     This field displays the element file name, preceded by the path to the element file. The naming convention is given in Table 13, on page 503.

     When you save the Template asset, the value in the **Element Storage Path/Filename** field is written to the `url` column of the `ElementCatalog` table, for the row that represents the element.

4. The **Rootelement** field is pre-populated with the value given in Table 13, on page 503. **Do not change the value of this field.** If the default value is changed, some tags that expect the default value, such as the `render:calltemplate` tag with the `style` attribute set to `element`, will fail.
5. (Optional). In the **ElementCatalog Description** field, type a description of the element. When you save the Template asset, information in this field is written to the `description` column for the element entry in the `ElementCatalog` table.
6. (Required). In the **Element Logic** field, code your element. Be sure to enter all of your code between the two `cs:ftcs` tags.

   If you are using JSP, remove the comments from the `taglib` directives that describe the tag libraries you are using.

   For help with this step, see Chapter 27, "Coding Elements for Templates and CSElements."

   **Note**

   **Ensuring Template Sharing or a Replicable Site**
   If you wish to share your Template asset or make the current site replicable, make sure that the template's element logic does not directly refer to assets, asset types, attribute names, or template names. Instead, use the `render:lookup` tag and prescribed keys as explained in "Template Sharing and Site Replication" on page 504. In "Step 5: Configure the Map" on page 517, you will map the same keys to the asset information that must be accessed for use in the element logic.

   **Calling a Template**
   Templates should ***always*** be called by the `render:calltemplate` tag, and never the `render:callelement` tag or `render:satellitepage` tag.

7. (Optional). The **Element Parameters** field and **Additional Element Parameters** field are used to enter variables or arguments that can be passed to the element, if the site design requires them.
   - The **Element Parameters** field corresponds to the `resdetails1` column in the `ElementCatalog`. When you save the template, CS-Direct writes the template ID (`tid`) to this field (i.e., to the `resdetails1` column).
   - The **Additional Element Parameters** field corresponds to the `resdetails2` column in the `ElementCatalog`. CS-Direct leaves this field blank.

   If your site design requires you to use variables in addition to `tid` in your template element, enter the variables into one of the fields above. Enter them as *`name`*=*`value`* pairs with multiple arguments separated by the ampersand (&) character. For example:

   ```
   MyArgument=value1&YourArgument=value2
   ```

   Each field supports up to 255 characters.

   For more information about using variables, see Chapter 4, "Programming with Content Server."
8. Click **Continue** to open the next screen, **SiteEntry**.

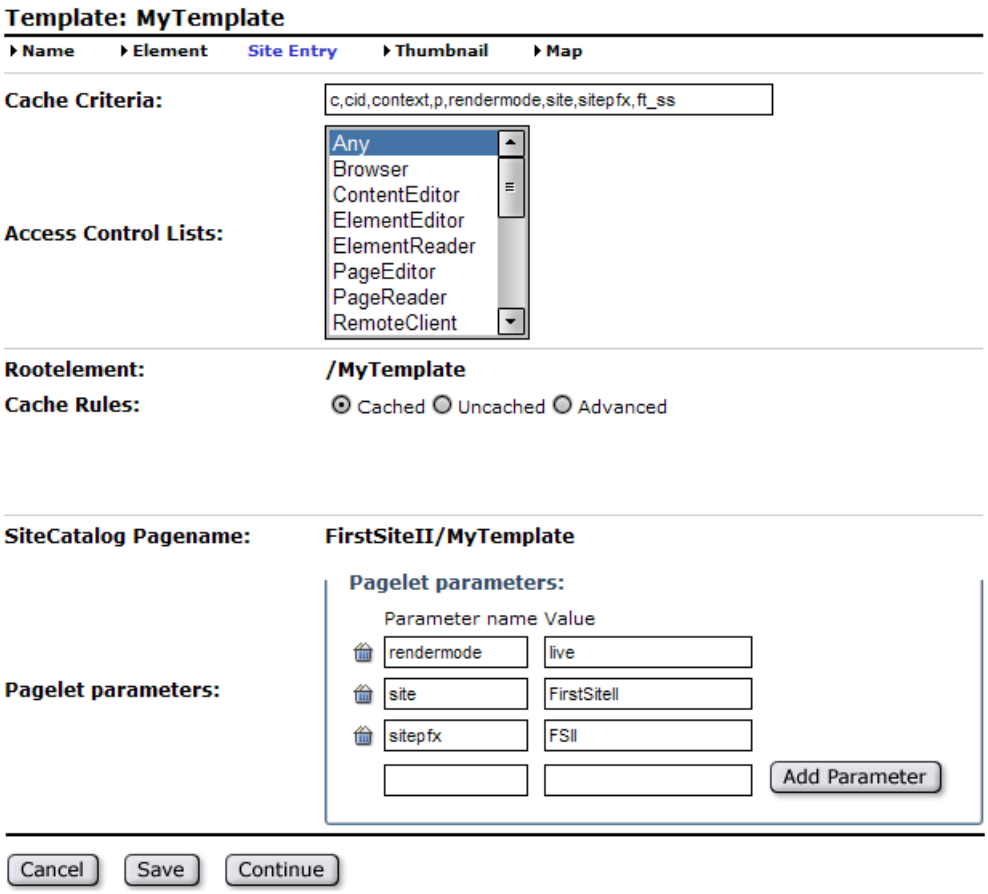
## Step 4: Configure SiteEntry

The **SiteEntry** screen is used to specify caching and pagelet parameters for the page to be rendered by this Template asset.

When the Template asset is saved, field values that you specify in the **SiteEntry** screen are written to the `SiteCatalog` table, as indicated in the procedures below.

### To configure the SiteEntry

1. In the **SiteEntry** screen, fill in the fields as explained in this section.




2. In the **Cache Criteria** field:
   a. CS-Direct names the following **reserved** variables as Cache Criteria:

   `c,cid,context,p,rendermode,site,sitepfx,ft_ss`

> **Note**
>
> The reserved Cache Criteria variables should not be removed. For information about the reserved variables, see Chapter 4, “Programming with Content Server.”

**b.** If you need to include your own variables as Cache Criteria (for example, **`foo`**), add them to the existing list. For example:

```
c,cid,context,foo,p,rendermode,site,sitepfx,ft_ss
```

> **Note**
>
> The **Cache Criteria** field names the variables which, in conjunction with SiteCatalog Pagename, define a pagelet as being unique. The variables are used to identify cached pages, which means that the variables are used in the page's cache key.
>
> Only those variables that are specified as Cache Criteria are used by the caching system to create the cache key for cached pages. Therefore, if your site design requires you to use page-level variables in addition to the reserved variables, be sure to designate them as Cache Criteria variables, as shown in this step.

**c.** If you need to enable the "Extra Parameters" section in FCKEditor, add the parameters to the existing list of Cache Criteria, using the `fp:`*`parameter_name`* naming convention. For example:

```
c,cid,context,p,rendermode,site,sitepfx,ft_ss,
   fp:imageHeight,fp:imageHeight
```

The "Extra Parameters" section provides a way of passing custom parameters, such as image dimensions, to the template. These extra parameters will be available in the **Include asset link** and **Add asset link** pop-up windows. The parameter names (`imageHeight` and `imageWidth` in our example) will be displayed in FCKEditor's "Extra Parameters" section, as options in the "Name" menu. The "Value" field enables the user to specify a value for the chosen parameter. For more information about extra parameters, see the Content Server Advanced and Dash user guides.

When the Template asset is saved, the Cache Criteria variables are written to the `pagecriteria` column in the `SiteCatalog` table.

**3.** (Optional). The **Access Control Lists** field corresponds to the `acl` column in the `SiteCatalog` table. If you want to allow only certain visitors to request this page, select the ACLs that the visitors must have in order to see the page. (For more information about ACLs, see Chapter 31, "User Management on the Delivery System.")

**4.** The **Rootelement** field is pre-populated with a value that is shown in Table 13, on page 503.

**5.** The **Cache Rules** field corresponds to the `cscacheinfo` and `sscacheinfo` columns in the `SiteCatalog` table. Do one of the following:

- Select **Cached** if the pagelet to be rendered by this template's element must be cached. The pagelet is set to be cached forever. The cache will be flushed by CacheManager's active cache management logic. This option sets both Content Server and Satellite Server caching conditions.
- Select **Uncached** if you wish to turn off caching for the pagelet to be rendered by this template's element. This option sets both Content Server and Satellite Server caching conditions.

- Select **Advanced** if you wish to set caching rules individually for Content Server and Satellite Server. Selecting **Advanced** displays two additional fields: one for Content Server caching and one for Satellite Server caching.

**Note**

CacheManager is designed to manage the lifecycle for cached pages on both Content Server and Satellite Server. It is designed to operate with pages that are set to be cached forever. If the cache expires on Content Server before it expires on Satellite Server, CacheManager will fail to flush the cache properly and invalid pages may be served from cache. Only advanced users should configure these settings manually.

For more information about page caching settings, see Chapter 5, "Page Design and Caching."

For more information about page caching settings, see Chapter 5, "Page Design and Caching."

6. **SiteCatalog Pagename** field. **Do not change the value of this field.** This field is pre-populated with the name of the page entry. The page naming convention is given in Table 13, on page 503.

7. In the **Pagelet parameters** section, you can enter pagelet parameters (name-value pairs), which will be passed into the template each time it is executed. (The **Pagelet parameters** section supports a total of 510 characters.)

**Note**

- The **Pagelet parameters** section is pre-populated with the following default pagelet parameters (reserved variables that were named in step 2a on page 513), including their values:

  ```
  site, sitepfx, rendermode
  ```

  The default parameter values will be overwritten if they are explicitly specified when the template is called.
- If you are specifying a pagelet parameter in this step, make sure to list its name as a Cache Criteria variable (see step 2 on page 513).
- If you named your own Cache Criteria variables (in step 2 on page 513), the variables are listed in the **Page parameters** section. If you do not specify values for these parameters, Content Server ignores the parameters.

When the Template asset is saved, the name-value pairs that are specified as **Pagelet parameters** are written to either the `resargs1` or `resargs2` column of the `SiteCatalog` table. The column to which they are written is not important and is managed automatically. (Each column supports up to 255 characters.)

8. Click **Continue** to open the next screen, **Thumbnails**.

9. Click **Continue** to open the next screen, **Map**.

**Note**

You will return to the **Thumbnail** screen after you have completed creating the Template asset and saved the Template asset.

## Step 5: Configure the Map

The purpose of mapping is to enable site replication and the sharing of Template assets, as explained in "Template Sharing and Site Replication" on page 504.

**Note**

Skip this section if you are designing a non-replicable site.

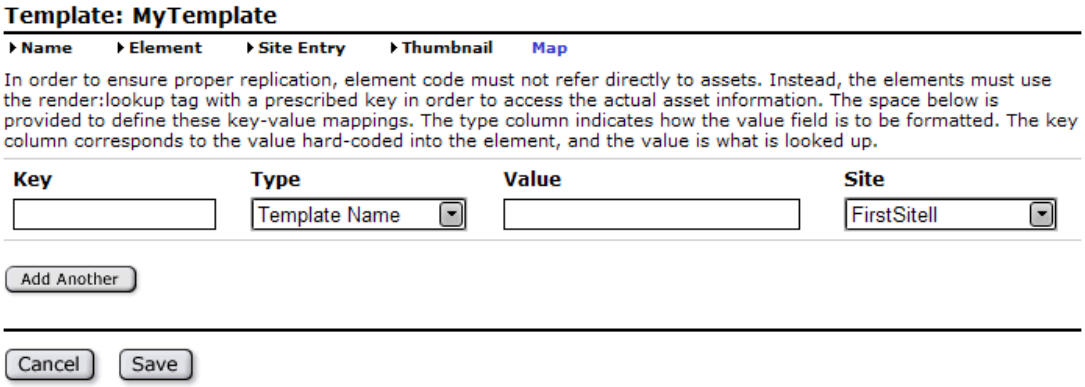
Using the **Map** screen, you will:

- Map each key in the `render:lookup` tags of the template's element logic to a value that will be used by the element logic.
- Map each key's value to the asset information that must be used in the element logic: asset, asset type, attribute name, or template name.

When the Template asset is saved, the map is written to the `Template_Map` table.

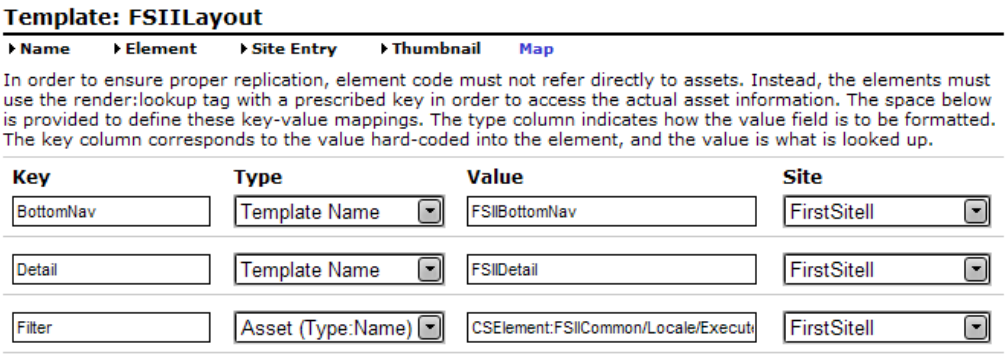
### To configure a map

In the **Map** screen, fill in the fields as explained in this section.




1. The **Key** field represents a value that the element logic will look up. Enter the key that is named in a `render:lookup` tag of the element logic.
2. The **Type** field identifies the type of asset information to be accessed. Select one of the following options:
   - **Template Name**—Maps a template name to the key value (which you will specify in the **Value** field, in the next step). The information that will be accessed is a template name that matches the value that you will specify in the next step. (For an example, see Figure 10.)
   - **Asset Type**—Maps an asset type to the key value. The information that will be accessed is an asset type, equal to the value that you will specify in the next step.
   - **Asset (Type:Name)**—Maps an attribute type:name to the key value. The information that will be accessed is an asset whose type and name match the value that you will specify in the next step.
   - **Asset (Type:ID)**—Maps an attribute type:ID to the key value. The information that will be accessed is an asset whose type and name match the value that you will specify in the next step.

**Figure 10:** Template Asset: Sample Map




3. In the **Value** field, enter a value for the key. This value will be looked up by the element logic when the Template is executed.
4. In the **siteid** field, select the name of the site to which the mapping applies.
5. To add a key, click **Add Another** and repeat the steps in this section.
6. When you have completed creating your template, save the template (click **Save Changes**).

   Content Server displays the template's **Inspect** form.
7. If you wish to create a thumbnail for your template, continue with "Step 6: (Optional) Create a Thumbnail." Otherwise, skip to "Step 7: Inspect the Template" on page 519.

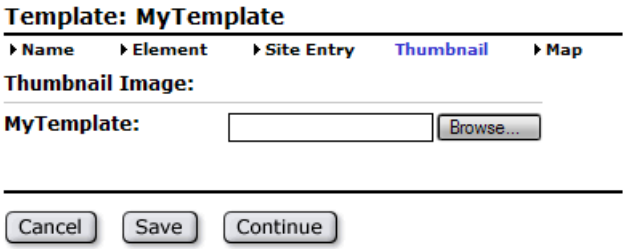
## Step 6: (Optional) Create a Thumbnail

A thumbnail graphically assists template users in determining how your Template asset lays out pages or renders content. The thumbnail that you create will be displayed in the Template's **Inspect** form.

When the Template asset is saved, the name of the thumbnail file is written to the `urlthumbnail` column of the `Template_Thumb` table.

### To create a thumbnail

1. Preview your Template asset. (For instructions, see "Templates and Preview" on page 539.)
2. Capture the preview as an image file and save it to a file system.
3. Open the **Template** form and click **Thumbnail** at the top of the screen.
4. In the **Thumbnail Image** field, enter (or browse for) the path to the image file that you created in step 2.



5. To display the thumbnail in the **Inspect** form:
   a. Save the template (click **Save Changes**).

      Content Server uploads the image file to the Content Server database and displays template's **Inspect** form.
   b. In the **Inspect** form, scroll down to the **Thumbnail Image** section. If the displayed image is too large or too small, resize the image in its source file and repeat steps 4 and 5.
6. To operate in the image in the **Thumbnail** screen:
   a. Scroll to the top of the **Inspect** form, and click the **Edit** link.
   b. At the top of the **Template** form, click **Thumbnail**.
7. To copy, send, and perform other operations on the thumbnail, right-click on the thumbnail and select an option.
8. If you wish to delete the thumbnail, select **Delete thumbnail image?** and click **Save Changes**.

   Content Server displays the template's **Inspect** form.

## Step 7: Inspect the Template

When you have finished creating the Template asset and clicked **Save**, CS-Direct does the following:

- Writes to the database tables:
  - Creates a template entry in the `Template` table.
  - Creates an element entry in the `ElementCatalog` table, using the *`AssetTypeName/TemplateName`* naming convention. If the element was coded in the template form (rather than CS-Explorer), CS-Direct also creates the element file.
  - Determines the name of the site that the template belongs to and creates a page entry in the `SiteCatalog` table using the *`SiteName/AssetTypeName/TemplateName`* naming convention.
  - Sets the name of the root element of the new `SiteCatalog` page entry to the name of the `ElementCatalog` entry.
  - Creates a thumbnail entry in the `Template_Thumb` table.
  - Creates a map entry in the `Template_Map` table.
- Displays the **Inspect** form (Figure 11, on page 520), which provides the following kinds of information:
  - Information in the **Name** screen (standard summary information, such as asset name, description, status, source, and ID, for assets of all types).
  - Information in the **Element** screen (root element, element logic, path to the element file, and `tid)`.
  - Information in the **SiteEntry** screen (SiteCatalog pagename, pagelet parameters, cache criteria, and the ACLs of users who are authorized to view the page).
  - Information in the **Thumbnail** screen (a thumbnail image, if one was chosen).

- Information in the **Map** screen (a map of key-value-asset information, if the site was designed to be replicable, or the template is sharable).
- If you have shared the Template asset, the **Inspect** form also lists all of the additional page entries in the `SiteCatalog` for this Template asset—there is a page entry for each site that the template is shared with.

**Figure 11:** Template Asset: Sample **Inspect** form

**Template: FSIISummary**

Inspect | Edit | Delete | more... | Add to My Active List

**Name:** FSIISummary
**Description:** Calls the Summary template directly, with no surrounding layout. Used primarily for preview, demo, and debugging.
**Status:** Created

**Locale:**

**Source:** Unavailable
**ID:** 1121304726689
**Category:** Page Template

**Usage:** Element defines a whole HTML page and can be called externally.
**For Asset Type:** Content, Content Parent, Document, Document Parent, Media, Media Parent, Page, Product, Product Parent, Query, Recommendation, Site
**Applies to subtypes:** *Any*

**Rootelement:** /FSIISummary
**ElementCatalog Description:**
**Element Storage Path/Filename:** FSIISummary.jsp
**Element Parameters:** tid=1121304726689
**Additional Element Parameters:**

**Element Logic:**

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld"
%><%@ taglib prefix="ics" uri="futuretense_cs/ics.tld"
%><%@ taglib prefix="render" uri="futuretense_cs/render.tld"
%><%@ taglib prefix="asset" uri="futuretense_cs/asset.tld"
%><%@ taglib prefix="string" uri="futuretense_cs/string.tld"
%><cs:ftcs><ics:if condition='<%=ics.GetVar("tid")!=null%>'><ics:then><render:logdep cid='<%=ics.GetVar("tid")%>'
<%-- This template is simply a typeless template that dispatches to the Summary
     template for the specified asset.  It is used primarily for previewing,
     demonstration, and debugging. --%>
<render:lookup site='<%=ics.GetVar("site")%>' varname="SummaryVar" key="Summary" tid='<%=ics.GetVar("tid")%>'/>
<render:calltemplate context="" tname='<%=ics.GetVar("SummaryVar")%>'
        site='<%=ics.GetVar("site")%>'
        tid='<%=ics.GetVar("tid")%>'
        slotname="SummaryTypeless"
        c='<%=ics.GetVar("c")%>'
        cid='<%=ics.GetVar("cid")%>'
        ttype="Template">
    <render:argument name="p" value='<%=ics.GetVar("p")%>'/>
    <render:argument name="locale" value='<%=ics.GetVar("locale")%>'/>
</render:calltemplate>
</cs:ftcs>
```

**Site Entries:**

| SiteCatalog Pagename | Pagelet parameters | Cache Criteria | ACL | Cache Rules |
|---|---|---|---|---|
| FirstSiteII/FSIISummary | rendermode : live<br>site : FirstSiteII<br>sitepfx : FSII | c, cid, context, locale, p, rendermode, site, sitepfx, ft_ss | *Any* | true,~0 / true,~0 |

**Map:**

| Key | Type | Value |
|---|---|---|
| Summary | Template Name | FSIISummary |

**Keywords:**

**Modified:** Jul 14, 2005 12:33:09 AM by firstsite

# Creating CSElement Assets

When you create a CSElement asset, you do three things: you create an asset, you code an element for the asset, and you configure a key-value-asset information map (similar to the map for a Template asset).

To create a CSElement asset, you must first complete the section "Prerequisites" (on this page) to determine how you will set CSElement properties that cannot (or must not) be changed once the CSElement is saved, and how you will code the CSElement's element logic. You will then complete the following steps, using the Content Server interface:

Step 1: Open the 'CSElement' Form

Step 2: Name and Describe the CSElement Asset

Step 3: Configure the Element – To specify its file type and logic

Step 4: Configure the Map – If you wish to support CSElement sharing and site replication

Step 5: Save and Inspect the CSElement

Step 6: Add the CSElement to the Active List—If you plan to use the CSElement as a root element for a Site Entry asset (see "Creating SiteEntry Assets" on page 531)

Information that you enter into the **CSElement** form will be written to database tables when the CSElement asset is saved, as indicated in the procedures below.

**Note**

Do not create CSElement assets directly in the database tables. Doing so will require you to write to several tables and can result in incorrect tracking of dependencies. Instead, use the **CSElement** form and the procedures in this section to create CSElement assets. For help with coding the CSElement logic, see Chapter 27, "Coding Elements for Templates and CSElements."

## Prerequisites

Before you begin creating a CSElement asset, you must determine several things:

- A name for your CSElement asset.
- Whether your CSElement will be sharable and the site replicable. These considerations determine how you will code the CSElement's element logic.
- Whether you plan to code the CSElement's element logic in CS-Explorer instead of the **CSElement** form. (This approach is not recommended for the reasons outlined in "Using Content Server Explorer to Create and Edit Element Logic" on page 540.)

### Naming the CSElement

- Once the CSElement asset is saved, its name cannot be changed.
- The CSElement logic file takes the name of the CSElement (followed by the file extension:

  `CSElementName.xml_or_jsp_or_html`

  The name of the CSElement logic file must not be changed.

## CSElement Sharing and Site Replication

Before creating a CSElement, decide whether the CSElement must be shared or the site you are working in must be replicable. If so, the CSElement logic will be coded in the same way. If sharing and replication are not required, you will skip key-value mapping ("Step 4: Configure the Map," on page 527).

For information about coding element logic to support CSElement sharing and site replication, see "Template Sharing and Site Replication" on page 504. The information applies without exception to CSElement assets.

# Procedures for Creating CSElement Assets

This section shows you how to create a CSElement asset, using the Content Server interface.

**Note**

Before starting the procedures in this section, read "Prerequisites" on page 521 for information about creating CSElement assets.

## Step 1: Open the 'CSElement' Form

1. Log in to the Content Server interface.
2. Select the site in which you want to work.
3. In the button bar, click **New**.
4. In the list of asset types, select **New CSElement**.

   **Note**

   For the **New CSElement** option to be displayed, the CSElement asset type must be enabled for your site and a start menu item must be created for it.

5. The **CSElement** form appears. Continue with "Step 2: Name and Describe the CSElement Asset."

   **Note**

   If you see a **Choose Assignees** screen instead of the **CSElement** form, it means that the CSElement you will be creating is associated with a workflow. Select a name (or names) from the "Users" column and click **Set Assignees**. Continue with "Step 2: Name and Describe the CSElement Asset."

## Step 2: Name and Describe the CSElement Asset

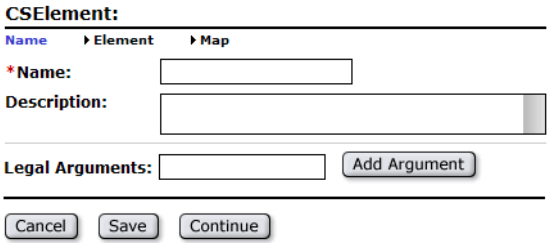
The **Name** screen is used to define metadata about the CSElement. From this metadata, a developer will be able to identify what the CSElement does and the arguments it uses to perform its function.

**Note**

At any time in the process of creating a CSElement, you can save the CSElement. Content Server will display the CSElement's **Inspect** form. To return to the **CSElement** form, click the **Edit** link.

### To name and describe the CSElement

1. In the **Name** screen, fill in the fields as explained in this section.




2. (Required). In the **Name** field, type a unique, descriptive name for the CSElement asset. It's best to use a name that describes what the CSElement does.

   **Valid entries:**

   - Up to 64 alphanumeric characters (the first character must be a letter)
   - Underscores (_)
   - Hyphens (-)
   - Spaces (these will be converted to underscores when used in the `SiteCatalog` pagename for the template)

   **Note**

   Make sure you have chosen a name for your CSElement asset using the guidelines in the section "Prerequisites" on page 531.

3. In the **Description** field, type a brief description of the CSElement asset. You can enter up to 128 characters.
4. In the **Legal Arguments** field:
   a. Enter an argument that may be passed to the CSElement and click **Add Argument**.
   b. In the fields that are displayed:
      - Specify whether the argument is optional or required.
      - Provide a description of the argument (to help you know the purpose of the argument you are creating).
      - Specify legal values (including descriptions) for the argument.

   (You can specify as many arguments and legal values as you require by clicking the **Add Arguments** and **Add Legal Value** buttons.)
5. Click **Continue** to open the next screen, **Element**.

## Step 3: Configure the Element

The **Element** screen (page 525) is used to create the CSElement's element—define the element file type (XML, JSP, or HTML), provide the element logic, and name the element. For example:

- The **Create Element** field offers a choice of XML, JSP, or HTML file types for the element logic, and is used to seed the **Element Logic** field with standard stub code (which you need to include in any element that you create).
- When you use the **Create Element** field to create, for example, a `.jsp` file, CS-Direct adds JSP `taglib` statements and the `render.logdep` tag to the **Element Logic** field by default so that the compositional dependency between this CSElement asset and pages that are rendered from this element is logged. For other file types, CS-Direct adds code specific to the file type. You will add your own code to the **Element Logic** field.

  For information about dependencies, see "About Dependencies" on page 580. For help with coding the element logic, see Chapter 27, "Coding Elements for Templates and CSElements."
- The **Element Storage Path/Filename** field names the file that holds the element logic and specifies the path to the file.

When the CSElement is saved, field values in the **Element** screen are written to a row (representing the element) in the `ElementCatalog` table, as indicated in the procedures below.

### Selecting an Existing Element

In the steps that follow, we assume you are creating a new element for the CSElement asset. If, however, you are migrating assets from an earlier Content Server release and wish to reuse an existing element, you need to identify the element correctly so that CS-Direct can find it and associate it with the CSElement asset.

**To select an existing element**

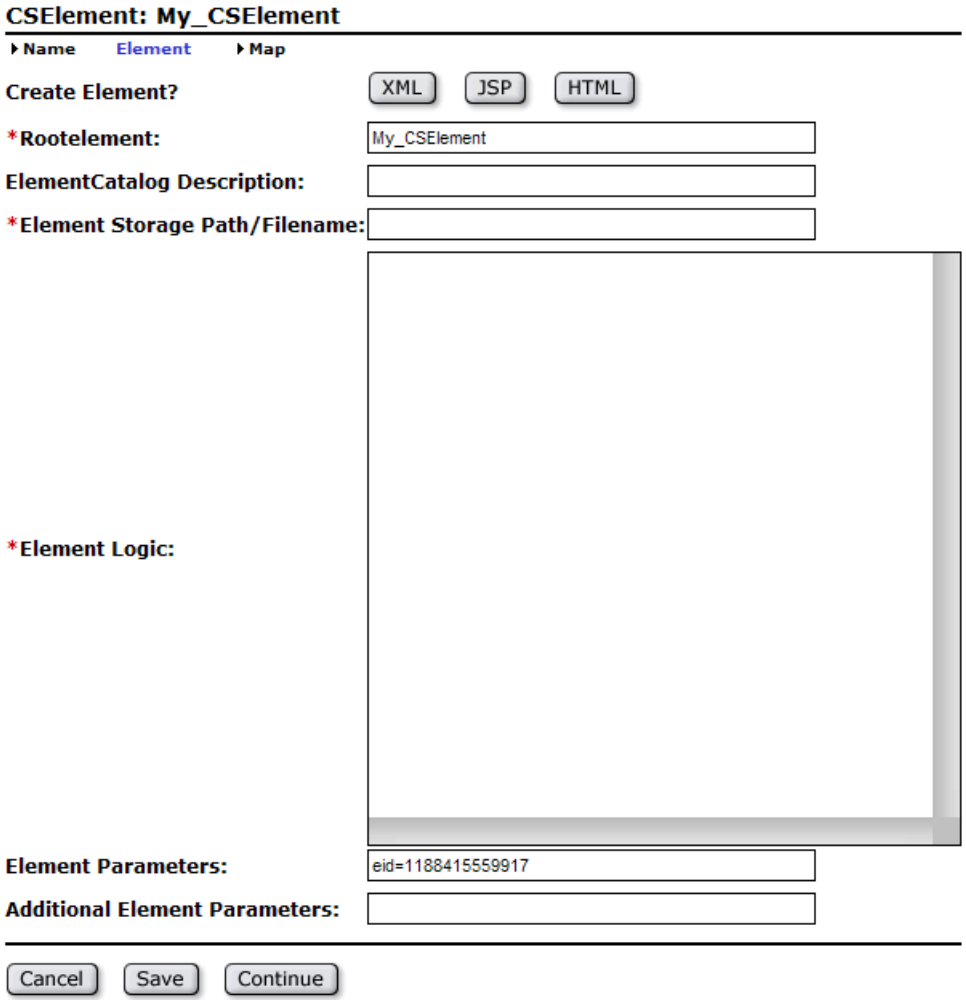
1. (Optional). In the **ElementCatalog Description** field, type a description of the element.
2. In the **Element Storage Path/Filename** field, enter a value according to the convention in "Naming the CSElement," on page 521.
3. If your site design requires it, enter the appropriate arguments in the element parameter fields. For instructions, see step 6 on page 526.
4. Save and re-open the CSElement asset.

   CS-Direct checks for the presence of an element with the correct name:

   - If the element has been correctly named, CS-Direct recognizes the element and displays its code in the **Element Logic** field.
   - If the named element does not exist (or is incorrectly named), CS-Direct does nothing. When you inspect or edit the CSElement asset, CS-Direct displays a message stating that there is no root element in the form. As soon as you code the element and give it the correct name, CS-Direct detects it and associates it with the CSElement asset.

### To configure a new element

1. In the **Element** screen, fill in the fields as explained in this section.




2. In the **Create Element** field, do one of the following:
   - To create an .xml file, click **XML**. The code that is pasted in comes from the `OpenMarket\Xcelerate\AssetType\CSElement\modelXML.xml` element and can be modified to use custom default logic.
   - To create a .jsp file, click **JSP**. The code that is pasted in comes from the `OpenMarket\Xcelerate\AssetType\CSElement\modelJSP.xml` element and can be modified to use custom default logic.
   - To create an .html file, click **HTML**. The code that is pasted in comes from the `OpenMarket\Xcelerate\AssetType\CSElement\modelHTML.xml` element and can be modified to use custom default logic.

   CS-Direct populates the following fields:
   - **Element Storage Path/Filename** field. **Do not change the value of this field.**

This field displays the element file name preceded by the path to the element file. By default, the file takes the name of the CSElement asset (entered in step 2 on page 523) followed by the file extension:

*CSElementName.xml_or_jsp_or_html*

When you save the CSElement asset, the value in this field is written to the `url` column of the `ElementCatalog` table, for the row that represents the element.

- **Element Logic** field with a header and other information.

  For example, if you clicked the **JSP** button, CS-Direct sets a tag library directive for some common CS-Direct JSP tag libraries (`asset`, `siteplan`, `render`). CS-Direct also sets the beginning and ending `cs:ftcs` tags, and a `RENDER.LOGDEP` (`render:logdep`) tag to mark a compositional dependency between the CSElement asset and any page or pagelet rendered by the element.

3. The **Rootelement** field is pre-populated with the name of the element file (*CSElementName.xml_or_jsp_or_html*). **Do not change the value of this field.**

   The root element is listed by this name in the `ElementCatalog` table. When you create code that calls this element (`RENDER.CALLELEMENT`), this is the name you should use. It uses the name of the CSElement asset by default.

4. (Optional). In the **ElementCatalogDescription** field, type a description of the element.

   When you save the CSElement asset, information in this field is written to the `description` column for the element entry in the `ElementCatalog` table.

5. (Required). In the **Element Logic** field, code your element. Be sure to enter all of your code before the ending `cs:ftcs` tag.

   If you are using JSP, remove the comments from the `taglib` directives that describe the tag families you are using.

   For help with this step, see Chapter 27, "Coding Elements for Templates and CSElements."

   **Note**

   **Ensuring Template Sharing or a Replicable Site**
   If you wish to share your CSElement or make the current site replicable, make sure that the CSElement's element logic does not directly refer to assets, asset types, attribute names, or template names. Instead, use the `render:lookup` tag and prescribed keys as explained in "Template Sharing and Site Replication" on page 504. In "Step 4: Configure the Map" on page 527, you will map the keys to the asset information that must be accessed for use in the element logic.

   **Calling a Template**
   Templates should ***always*** be called by the `render:calltemplate` tag, and never the `render:callelement` tag or `render:satellitepage` tag.

6. (Optional). The **Element Parameters** field and **Additional Element Parameters** field are used to enter variables or arguments that can be passed to the element, if the site design requires them.

- **Element parameters** field. CS-Direct populates this field with the CSElement ID (`eid`), generated by Content Server as a unique identifier of the CSElement asset. **Do not change or delete this value**.

  This field corresponds to the `resdetails1` column of the `ElementCatalog` table. When you save the CSElement, CS-Direct writes the CSElement ID to the `resdetails1` column, in the row that represents the CSElement.

- **Additional element parameters** field. CS-Direct leaves this field blank.

  This field corresponds to the `resdetails2` column of the `ElementCatalog`.

If your site design requires you to use variables in addition to `eid`, enter the variables into one of the fields above. Enter them as *`name`*=*`value`* pairs with multiple arguments separated by the ampersand (&) character. For example:

```
MyArgument=value1&YourArgument=value2
```

Each field supports up to 255 characters.

For more information about Content Server variables, including scope and precedence, see Chapter 4, "Programming with Content Server."

7. Click **Continue** to open the next screen, **Map**.

## Step 4: Configure the Map

The purpose of mapping is to enable site replication and sharing of CSElement assets. The concepts behind mapping are identical to those for Template assets. They are explained in "Template Sharing and Site Replication" on page 504.

**Note**

Skip this section if you are designing a non-replicable site or a CSElement asset that will not be shared.

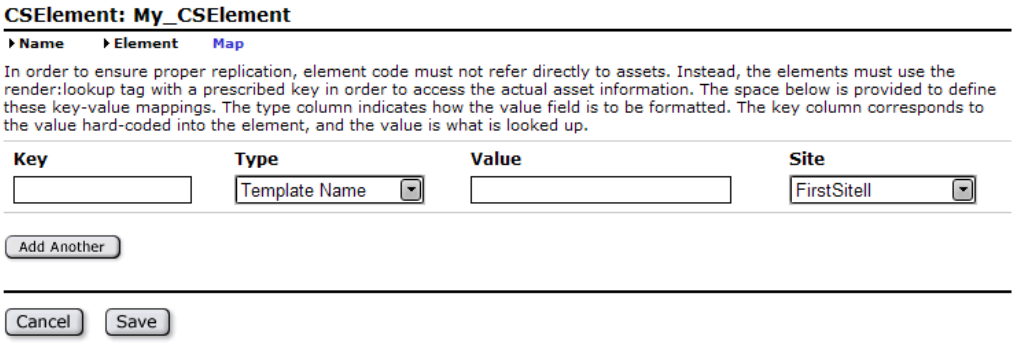
Using the **Map** screen, you will:

- Map each key in the `render:lookup` tag of the element logic to the value that must be used in the element logic.
- Map each key's value to the asset information that must be used in the element logic: asset, asset type, attribute name, or template name.

When the CSElement asset is saved, the map is written to the `CSElement_Map` table.

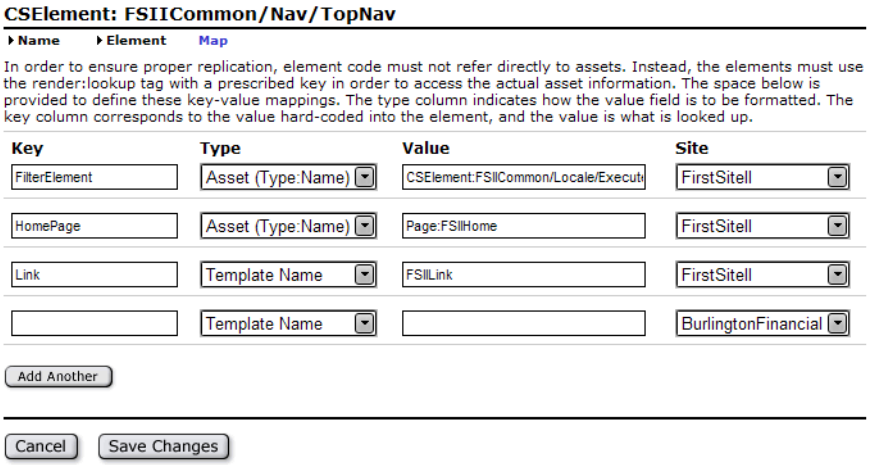
### To configure a map

1. In the **Map** screen, fill in the fields as explained in this section.




2. The **Key** field represents the value that the element logic will look up. In this field, enter the key that is named in a `render:lookup` tag of the element logic.
3. The **Type** field identifies the type of asset information to be accessed. Select one of the following options:
   - **Template Name**—Maps a template name to the key value (which you will specify in the **Value** field, in the next step). The information that will be accessed is a template name that matches the value you will specify in the next step. (For an example, see Figure 12, on page 528.)
   - **Asset Type**—Maps an asset type to the key value. The information that will be accessed is an asset type, equal to the value that you will specify in the next step.
   - **Asset (Type:Name)**—Maps an attribute type:name to the key value. The information that will be accessed is an asset whose type and name match the value that you will specify in the next step.
   - **Asset (Type:ID)**—Maps an attribute type:ID to the key value. The information that will be accessed is an asset whose type and name match the value that you will specify in the next step.

**Figure 12:** CSElement Asset: Sample Map



4. In the **Value** field, enter a value for the key. This value will be looked up by the element logic when the CSElement asset is invoked.
5. In the **siteid** field, select the name of the site to which the mapping applies.
6. To add a key, click **Add Another** and repeat the steps in this section.

## Step 5: Save and Inspect the CSElement

When you have finished creating the CSElement asset, click **Save**.

CS-Direct does the following:

- Writes to the database tables:
  - Creates a row in the `CSElement` table for the CSElement asset, where it enters the CSElement name and description that you specified in the steps above.
  - Creates an element entry in the `ElementCatalog` table using values specified in the **Element** screen:
    - The value of the **Rootelement** field is used to position the element file in the appropriate folder.
    - The value of the **Element Storage Path/Filename** field is written to the `url` column.
    - The value of the `eid` variable is set to the ID of the CSElement asset in the `resdetails1` column.
- Flushes the pagecache of any pagelets that call this element.
- Displays the **Inspect** form (Figure 11, on page 520), which provides the following kinds of information:
  - Information in the **Name** screen (standard summary information, such as asset name, description, status, and ID, for assets of all types).
  - Information in the **Element** screen (root element, element logic, path to the element file, and the element's `eid`).
  - Information in the **Map** screen (a map of key-value-asset information, if the site was designed to be replicable, or the template is sharable).
  - **Preview with Arguments** button, enabling you to preview the page(s) rendered by the SiteEntry asset.

## Step 6: Add the CSElement to the Active List

**Note**

Complete the steps in this section if you are planning to use your CSElement to create a SiteEntry asset. This step makes the CSElement available for selection in Content Server's tree by adding it to your active list.

If you are not planning to create the SiteEntry asset in this session, you might want to add the CSElement to your active list so that you can easily find it later.

### To add the CSElement to the Active List

1. Run a search on the CSElement asset you created.

2. In the results list, select the checkbox in the right-hand column to add the CSElement to the **Active List**.

   This CSElement is listed in Content Server's tree, in the **Active List** tab, where it is a selectable option for SiteEntry assets.

3. Create the SiteEntry asset. For instructions, see "Creating SiteEntry Assets" on page 531.

**Figure 13:** CSElement Asset: Sample **Inspect** Form

**CSElement: FSIICommon/Login/LoginUser**

Preview | Inspect | Edit | Delete | more... | Add to My Active List

| | |
|---|---|
| **Name:** | FSIICommon/Login/LoginUser |
| **Description:** | Login FSII User |
| **Status:** | Created |
| **Locale:** | |
| **ID:** | 1121172250311 |
| **Rootelement:** | FSIICommon/Login/LoginUser |
| **ElementCatalog Description:** | |
| **Element Storage Path/Filename:** | FSIICommon/Login/LoginUser.jsp |
| **Element Parameters:** | eid=1121172250311 |
| **Additional Element Parameters:** | |

**Element Logic:**

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld" %>
<%@ taglib prefix="ics" uri="futuretense_cs/ics.tld" %>
<%@ taglib prefix="render" uri="futuretense_cs/render.tld" %>
<cs:ftcs>
    <ics:if condition='<%=ics.GetVar("seid")!=null%>'>
        <ics:then>
            <render:logdep cid='<%=ics.GetVar("seid")%>' c="SiteEntry"/>
        </ics:then>
    </ics:if>
    <ics:if condition='<%=ics.GetVar("eid")!=null%>'>
        <ics:then>
            <render:logdep cid='<%=ics.GetVar("eid")%>' c="CSElement"/>
        </ics:then>
    </ics:if>

    <%-- This element takes incoming ICS variables and creates the user
         based on that information.  If CS Engage is present (and the Engage Core
         module is installed) then it will also call an element that will
         create the visitor object.

         It requires the following input:
           - VisitorUserName
           - VisitorID
    --%>
    <ics:setssvar name="VisitorUserName" value='<%= ics.GetVar("VisitorUserName") %>'/>
    <ics:setssvar name="VisitorID" value='<%= ics.GetVar("VisitorID") %>'/>

    <%-- log the user into engage, if the engage core module is installed --%>
    <render:lookup site='<%=ics.GetVar("site")%>' match=":x" varname="LoginUserEngage" key=
    <ics:if condition='<%=ics.GetVar("LoginUserEngage") != null %>'>
        <ics:then>
            <render:callelement elementname='<%=ics.GetVar( "LoginUserEngage" ) %>'>
                <render:argument name="VisitorUserName" value='<%= ics.GetVar("VisitorUserN
            </render:callelement>
        </ics:then>
    </ics:if>
</cs:ftcs>
```

**Map:**

| Key | Type | Value |
|---|---|---|
| LoginUserEngage | Asset (Type:Name) | CSElement:FSIICommon/Login/LoginUserEngage |

**Preview with Arguments:** Preview...

**Modified:** Jul 12, 2005 9:00:29 AM by firstsite

# Creating SiteEntry Assets

When you create a SiteEntry asset, you are creating both an asset and a page entry in the `SiteCatalog` table. The fields in the first part of the **SiteEntry** form define the page entry as an asset. The rest of the fields provide information about the page entry as a Content Server page, information that is written to the `SiteCatalog` table.

To create a SiteEntry asset, you must first complete the section "Prerequisites" (on this page) to create its root element and determine how you will set SiteEntry properties (such as SiteEntry name). You will then complete the following steps, using the Content Server interface:

Step 1: Open the 'SiteEntry' Form

Step 2: Create the SiteEntry Asset

Step 3. Save and Inspect the SiteEntry Asset

Information that you enter into the **SiteEntry** form will be written to database tables when the CSElement asset is saved, as indicated in the procedures below.

**Note**

Do not create SiteEntry assets directly in the database tables. Doing so will require you to write to several tables and can result in incorrect tracking of dependencies. Instead, use the **SiteEntry** form and the procedures in this section to create SiteEntry assets.

## Prerequisites

Before you begin creating a SiteEntry asset, complete the following steps:

- Create a root element for the page entry:
  1. Create a CSElement asset. For instructions, see "Creating CSElement Assets" on page 521.

     (A root element is required for any page entry. The root element is the element of the CSElement, which you will select for the SiteEntry asset.)
  2. Make the CSElement available. For instructions, see "Step 6: Add the CSElement to the Active List" on page 529.

     (To specify a CSElement for your SiteEntry asset, you will select the CSElement from the **Active List** tab in Content Server's tree, or the **History** tab if you created the CSElement in the current session).
- Determine a name for your SiteEntry asset. The same name will be assigned to the page. Neither the SiteEntry name nor the page name can be changed once the SiteEntry asset is saved.

## Procedures for Creating SiteEntry Assets

This section shows you how to create a SiteEntry asset, using the Content Server interface.

**Note**

Before starting the procedures in this section, read "Prerequisites" on page 531 for information about creating SiteEntry assets.

### Step 1: Open the 'SiteEntry' Form

1. Log in to the Content Server interface.
2. Select the site in which you want to work.
3. In the button bar, click **New**.
4. In the list of asset types, select **New SiteEntry**.

   **Note**

   For the **New SiteEntry** option to be displayed, the SiteEntry asset type must be enabled for your site and there must be a start menu item created for it.

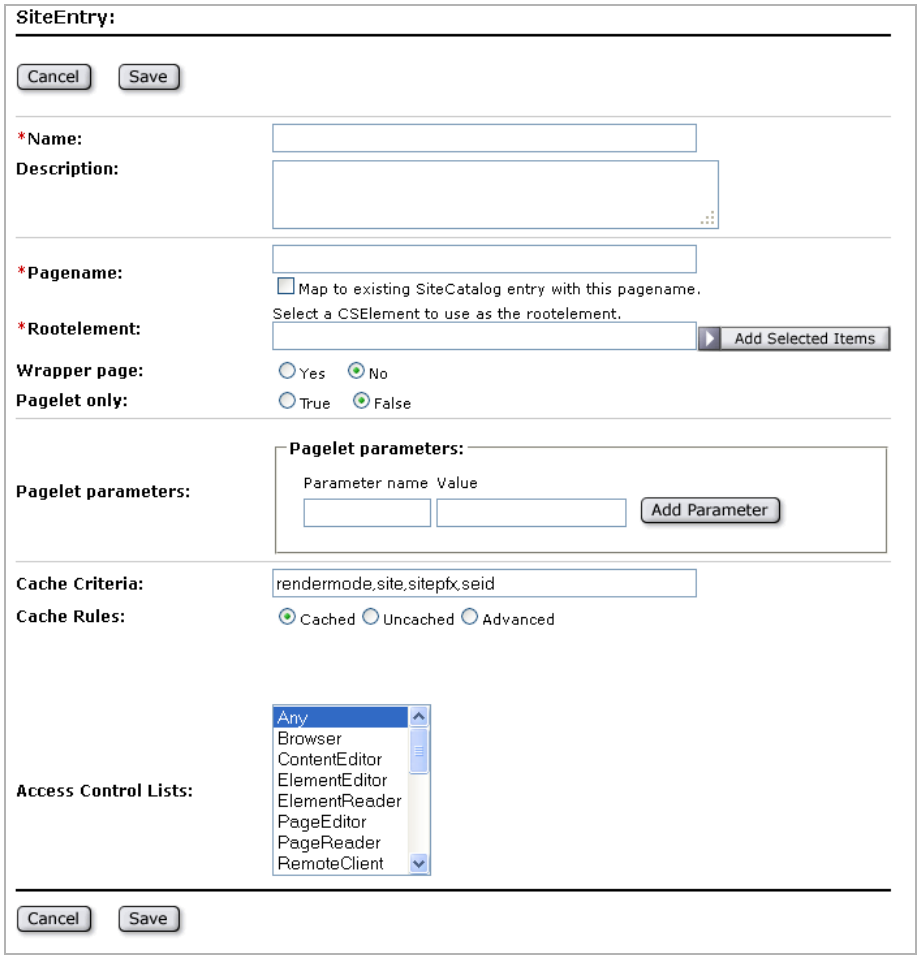
5. The **SiteEntry** form appears. Continue with "Step 2: Name and Describe the CSElement Asset."

   **Note**

   If you see a **Choose Assignees** screen instead of the **SiteEntry** form, it means that the SiteEntry you will be creating is associated with a workflow. Select a name (or names) from the "Users" column and click **Set Assignees**. Continue with "Step 2: Name and Describe the CSElement Asset."

## Step 2: Create the SiteEntry Asset

1. In the **SiteEntry** form, fill in the fields as explained in this section.




2. (Required). In the **Name** field, type a descriptive name for the SiteEntry asset. It's best to use a name that describes the purpose of the page.

   **Valid entries:**

   - Up to 64 alphanumeric characters (the first character must be a letter)
   - Underscores (_)
   - Hyphens (-)
   - Spaces (these will be converted to underscores when used in the `SiteCatalog` pagename for the template).

3. In the **Description** field, type a brief description of the SiteEntry asset. You can enter up to 128 characters.

4. (Required). Click in the **Pagename** field to automatically populate it with the name of the page entry and the path to the page entry (for example: `FSIICommon/SideNav/ProductView`). **Do not change the value of this field.**

> **Note**
>
> The value in this field is the name of the page entry (which will be stored in the `SiteCatalog` table when the SiteEntry is saved). When you create code that calls this SiteEntry asset (`RENDER.SATELLITEPAGE`), this is the name you should use.

5. If you wish to use the row of a pre-existing page entry in the `SiteCatalog` table, select the **Map to existing SiteCatalog entry** checkbox.
6. (Required). In the **Rootelement** field, select the appropriate CSElement asset from the tree and click **Add Selected Items**.

> **Note**
>
> Only one CSElement can be added.

7. In the **Wrapper page** field, select one of the radio buttons to specify whether the asset you are creating is a wrapper page. Selecting the **NO** radio button displays the **Pagelet only** field.
8. If the **Pagelet only** field is displayed, select one of the radio buttons to specify whether the asset you are creating is a pagelet.
9. In the **Pagelet parameters** section, you can enter pagelet parameters (name-value pairs), which will be passed into the page or pagelet each time it is executed. (The **Pagelet parameters** section supports a total of 510 characters).

> **Note**
>
> - The **Pagelet parameters** section is pre-populated with the following default parameters (reserved variables that are named by default in the Cache Criteria field (next step), including their values:
>
>  `site, seid, sitepfx, rendermode`
>
>  The default values will be overwritten if they are explicitly specified when the page or pagelet is called.
> - If you are specifying a pagelet parameter in this step, make sure to list its name as a Cache Criteria variable in the next step.

When the SiteEntry asset is saved, the name-value pairs that are specified as **Pagelet parameters** are written to either the `resargs1` or `resargs2` column of the `SiteCatalog` table. The column to which they are written is not important and is managed automatically. (Each column supports up to 255 characters.)

10. In the **Cache Criteria** field:
    a. CS-Direct names the following **reserved** variables as Cache Criteria:

    `rendermode,seid,site,sitepfx,ft_ss`

**Note**

The reserved Cache Criteria variables should not be removed. For information about the reserved variables, see Chapter 4, "Programming with Content Server."

**b.** If you need to include your own variables as Cache Criteria (for example, **`foo`**), add them to the existing list. For example:

```
foo,rendermode,seid,site,sitepfx,ft_ss
```

**Note**

The **Cache Criteria** field names the variables which, in conjunction with Pagename, define a pagelet as being unique. The variables are used to identify cached pages, which means that the variables are used in the page's cache key.

Only those variables that are specified as Cache Criteria are used by the caching system to create the cache key for cached pages. Therefore, if your site design requires you to use page-level variables in addition to the reserved variables, be sure to designate them as Cache Criteria variables, as shown in this step.

When the SiteEntry asset is saved, Cache Criteria variables and their values are written to the `pagecriteria` column in the `SiteCatalog` table.

**11.** The **Cache Rules** field corresponds to the `cscacheinfo` and `sscacheinfo` columns in the `SiteCatalog` table. Do one of the following:

- Select **Cached** if the pagelet to be rendered by this SiteEntry's CSElement must be cached. The pagelet is set to be cached forever. The cache will be flushed by CacheManager's active cache management logic. This option sets both Content Server and Satellite Server caching conditions.
- Select **Uncached** if you wish to turn off caching for the pagelet to be rendered by this SiteEntry's CSElement. This option sets both Content Server and Satellite Server caching conditions.
- Select **Advanced** if you wish to set caching rules individually for Content Server and Satellite Server. Selecting **Advanced** displays two additional fields: one for Content Server caching and one for Satellite Server caching.

**Note**

CacheManager is designed to manage the lifecycle for cached pages on both Content Server and Satellite Server. It is designed to operate with pages that are set to be cached forever. If the cache expires on Content Server before it expires on Satellite Server, CacheManager will fail to flush the cache properly and invalid pages may be served from cache. Only advanced users should configure these settings manually.

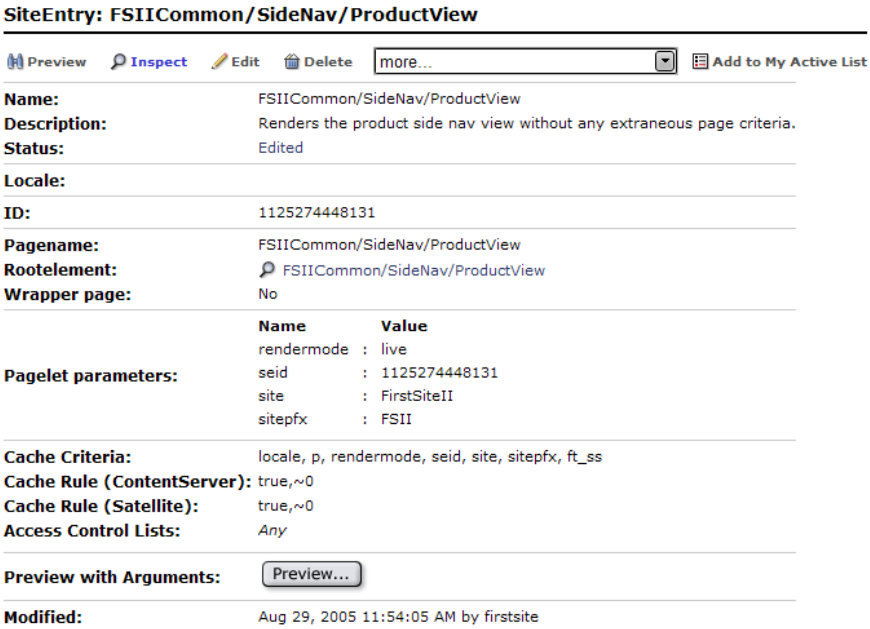
For more information about page caching settings, see Chapter 5, "Page Design and Caching."

12. The **Access Control Lists** field corresponds to the `acl` column in the `SiteCatalog` table. If you want to allow only certain visitors to request this page, enter the ACLs that visitors must have in order to see the page. (For more information about ACLs, see Chapter 31, "User Management on the Delivery System.")

## Step 3. Save and Inspect the SiteEntry Asset

When you have finished creating the SiteEntry asset, click **Save**. CS-Direct does the following:

- Writes to the database tables:
  - Creates a row in the `SiteCatalog` table for the SiteEntry asset, where it enters the values that you specified in the steps above.
- Displays the **Inspect** form, which provides the following information:
  - Standard summary information (asset name, description, status, ID) and the page entry criteria you specified in the steps above.
  - **Preview with Arguments** button, enabling you to preview the page(s) rendered by the SiteEntry asset.



# Managing Template, CSElement, and SiteEntry Assets

This section presents additional procedures for working with template, CSElement, and SiteEntry assets:

- Designating Default Approval Templates (Static Publishing Only)
- Editing Template, CSElement, and SiteEntry Assets
- Sharing Template, CSElement, and SiteEntry Assets
- Deleting Template, CSElement, and SiteEntry Assets
- Previewing Template, CSElement, and SiteEntry Assets

## Designating Default Approval Templates (Static Publishing Only)

When assets are approved for a publishing destination that uses the Export to Disk publishing method, the approval system examines the template assigned to the asset to determine its dependencies.

If you design your online site to render assets with more than one template (a text-only version and a summary version and a full version for the same type of asset, for example), you should create a template that contains a representative set of approval dependencies for all of the templates, and then specify that template as the Default Approval Template for the asset type.

For more information about approval templates, see "Approval Templates for Export to Disk" on page 583. For an example of a template that could be used as a default approval template, see the Burlington Financial template for article assets named Full.

### To designate that a template is the default approval template

1. On the **Admin** tab, select **Publishing > Destinations > Static**.
2. Under the name of a static destination, select **Set Default Templates**.
3. In the "Default Templates" form, click **Edit**.
4. In the edit form, select a default template for each asset type. If you are using the Subtype feature for any of your asset types, you can designate a default approval template for each subtype of that asset type. (For information about subtypes, see "Step 8: (Optional) Configure Subtypes" on page 295.)
5. When you have finished, click **Save**.

## Editing Template, CSElement, and SiteEntry Assets

Creating a template, CSElement, and SiteEntry assets also creates entries in the `SiteCatalog` and/or ElementCatalog tables. The names of those entries are based on the asset's name, and for Template assets, the asset type, and the site the template belongs to. Because these naming dependencies exist, the following restrictions apply when you edit templates, CSElements, or SiteEntry assets:

- You cannot rename a template, CSElement, or SiteEntry asset after it has been saved.
- For templates, you cannot change the asset type selected in the **Asset Type** field after the Template asset has been saved.

- For templates and CSElements, you cannot change the name of the root element.
- For SiteEntries, you cannot change the name of the page entry.

**Caution**

If you have manually created (using CS Explorer) one or more site entries that point to a template, and then edit the template through the Advanced interface, the manually created site entries are automatically deleted.

For the basic procedure for editing assets, see the *Content Server Advanced Interface User's Guide*.

## Sharing Template, CSElement, and SiteEntry Assets

When you share a CSElement, template, or SiteEntry asset, Content Server creates a row in the `AssetPublication` table for each site that you share the asset with.

Additionally, for **Template assets only**, CS-Direct does the following:

- Creates a new `SiteCatalog` page entry for each site that you share the asset with. It uses the name of the site in the name of the page entry. All of the new page entries point to the same root element, the template element.

  **Note**

  Do not change the root elements of these page entries. All page entries for a shared template must point to the same root element.

- Lists all the other page entries for the shared template that share this root element in the **Inspect** form.

For the basic procedure for sharing assets, see the *Content Server Advanced Interface User's Guide*.

**Note**

For templates and CSElements to be sharable, their element logic must not be hard-coded with asset type names, attribute names, template names, or IDs. Instead, use the `render:lookup` tag and hard-code the keys for which you have created a map that the `render:lookup` tag can refer to in order to look up asset information for use in the element logic.

## Deleting Template, CSElement, and SiteEntry Assets

CS-Direct does not allow you to delete an asset if there is another asset using it. However, it does not check to see whether a template or CSElement is referenced by the code in other template or CSElement elements.

Before you delete a template or SiteEntry asset, be sure to remove any page calls to that asset's page entry from your elements. Before you delete a CSElement asset, be sure to remove any element calls to that asset's root element from your other elements.

When you delete an asset, CS-Direct does the following:

- Changes the value of the asset's `name` column in the `Template`, `CSElement`, or `SiteEntry` table (depending on the asset type) to its object ID.
- Changes the value of the asset's `status` column in the `Template` table to `VO`, for "void".
- For templates, deletes all the `SiteCatalog` table entries (if the template is shared, there are as many page entries as there are sites that the template is shared with) and the `ElementCatalog` table entry for the template.
- For CSElements, deletes the `ElementCatalog` table entry for the asset.

For the basic procedure for deleting assets, see the *Content Server Advanced Interface User's Guide*.

## Previewing Template, CSElement, and SiteEntry Assets

Because template, CSElement, and SiteEntry assets provide logic and code for formatting other assets, you preview assets of these types differently from the way you preview your content assets.

### Templates and Preview

You preview a template by previewing an asset and selecting the template that you want to use to render the asset. Content Server invokes the code in the template and renders a page with the asset as the content.

### CSElement and SiteEntry Assets and Preview

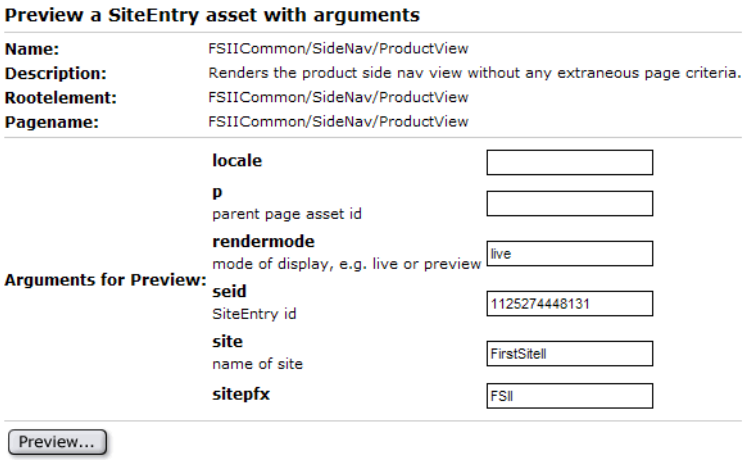
You preview CSElement and SiteEntry assets directly. If the element that will be called has self-contained context—a banner that does not expect variables or arguments, for example—you can simply click the **Preview** icon. But when the results of the rendered element depend on values that are passed to it, you must manually set those values in the CSElement or SiteEntry form in order to preview that asset.

For example, the Burlington Financial CSElement asset named `BurlingtonFinancial/Query/ShowHotTopics` expects a value for the `p` variable. If it doesn't receive one, the value of `p` defaults to the object ID of the Home page asset. If you want to preview this CSElement for a page asset other than the Home page, you must pass in the ID of that page asset as the value of the `p` variable with the argument fields in the "New" or "Edit" form for that CSElement asset.

#### To specify argument values for previewing CSElement or SiteEntry assets

1. Find the asset and inspect it (click the icon with the letter "i").

2. Scroll to the bottom of the **Inspect** form. Next to **Preview with Arguments**, click **Preview**. The following form appears for a CSElement asset:




3. Enter values for the arguments. You can also select values by double-clicking in the fields and selecting from the drop-down list.
4. Click **Preview**.
5. Click the links that are displayed to preview the pages that are rendered by this SiteEntry asset.

# Using Content Server Explorer to Create and Edit Element Logic

**Note**

FatWire does not recommend creating or editing element logic directly from CS-Explorer. However, should you prefer to do so, you will need to ensure the validity of your Template and CSElement assets. Take note of the information in this section and follow the instructions that are included.

When a Template (CSElement) asset is created and saved in the Content Server interface (**Template** or **CSElement** form), several important steps are taken that are not taken when you use CS-Explorer:

- The interface seeds your element with stub code that sets compositional dependencies and, if you are using JSP, drops in the appropriate tag library directives for you. (Compositional dependencies are described in the section "About Dependencies" on page 580.)
- When you save the Template (CSElement) in the Content Server interface:
  - The approval system receives information that the asset was changed and can therefore change its approval status.
  - Most importantly, the CacheManager servlet can update the cache (that is, flush pages and pagelets from the Content Server and Satellite Server caches).

If you choose to work with the CSElement asset in CS-Explorer, be sure that you do not alter the value of the `eid` variable or accidentally delete it.

A practical reason for using the Content Server interface is to avoid switching between Content Server and CS-Explorer, especially if you are mapping asset information (to support template sharing and site replication). Mapping is supported only in the Content Server interface (in the **Map** screen of the **Template** form and in the **Map** screen of the **CSElement** form).

## Creating Templates and CSElements

If you prefer to use CS-Explorer to code your element logic, follow the steps below:

1. Start creating your Template (or CSElement) asset using the **Template** form (or **CSElement** form). Start with "Step 1: Open the 'Template' Form" on page 505 (or "Step 1: Open the 'CSElement' Form" on page 522) and continue sequentially.
2. In "Step 3: Configure the Template's Element" on page 508 (or "Step 3: Configure the Element" on page 524), select your element type (JSP, XML, or HTML). Do not change the element logic that is auto-generated for you. Also, be sure that you do not alter the value of the `tid` variable or accidentally delete it.
3. Continue through the form you have chosen until you finish. If you know which keys and asset values you must map, add them to the **Map** form (in "Step 5: Configure the Map" on page 517. The same step applies to the CSElement asset).
4. Save the asset.
5. Open CS-Explorer and edit your element. Save your changes.
6. The final—and most important—step: Re-save your asset in the **Template** form (or CSElement form). You do not have to change any data in the form, but you must re-save it. This will ensure that no functionality is bypassed.

## Editing Templates and CSElements

Any time that you edit an element's logic in the CS-Explorer tool, open and save the template (or CSElement) in the Content Server interface so that (1) the approval system knows the asset was changed and can change its approval status, and (2) the CacheManager servlet can update the cache.

Chapter 24

# Creating Templates to Support Graphical Page Design

Content Server's InSite interface hosts the Page Layout utility for graphically creating and redesigning web pages. Page Layout complements the InSite Editing utility; whereas InSite Editing enables users to edit content directly on the rendered page, Page Layout is used to provide and position that content.

Using Page Layout requires developers to code templates that support graphical page design. This chapter outlines the concepts that are used by Page Layout and describes the implementation. This chapter contains the following sections:

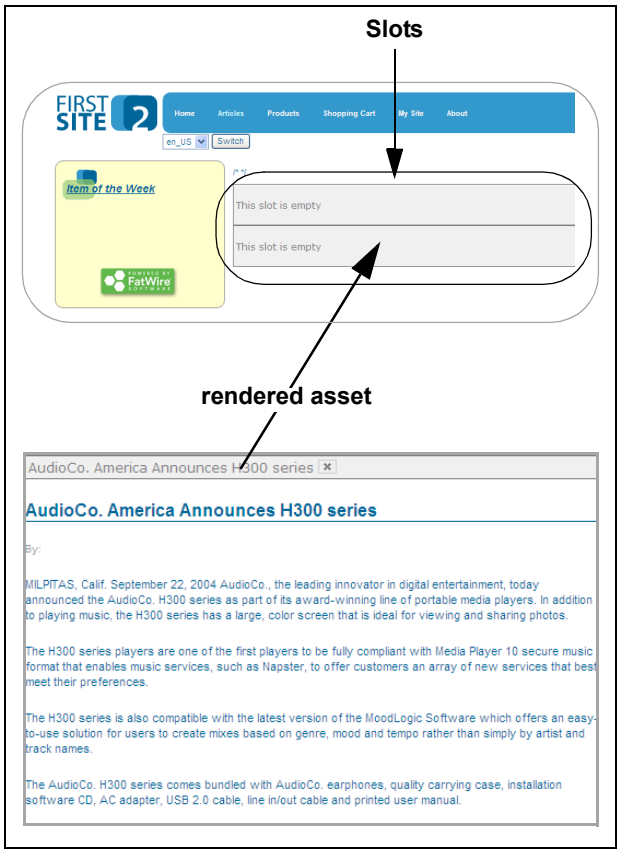

- Overview
- Implementation
- Template Context
- Tracking Changes to Master Pages

# Overview

Content Server's InSite interface hosts the Page Layout utility for graphically creating and redesigning web pages. Enabling Page Layout requires a template that displays slots into which content providers can then drag and drop rendered assets of their own choice.

Placed assets can be dragged from one slot into another, they can be replaced with other assets, or they can be deleted, without the user ever having to manipulate the underlying template code. Slots, however, cannot be repositioned on the page, except through template code.

Templates for the Page Layout utility are not out-of-the-box constructs. They must be coded to support slots in specified places on the pages they render.




# Implementation

For developers, creating slots means coding a JSP or XML template with as many `<insite:calltemplate>` tags as there are slots. The template is called a **master template** and the slotted pages it renders are called **master pages**.

**Note**

Master templates are indistinguishable from other templates. We call them "master templates" to indicate that they render pages with slots, and the slots are reusable — content which is dropped into a slot is not permanently fixed in the slot. As a result, a master template can be reused in multiple places on a site such that its slots simultaneously display different pieces of content. For a more detailed description of reusability, see "Template Context," on page 548.

The process of coding and using a master template is summarized in the steps below. (For detailed instructions on using the Page Layout utility, see the Content Server user's guides.)

1. A developer codes a master template. In this example, the developer creates two slots, using the `insite.tld` in line 5 and the `<insite:calltemplate>` tag in lines 18–24 and 27–33:

```
1   <%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld"%>
2   <%@ taglib prefix="ics" uri="futuretense_cs/ics.tld"%>
3   <%@ taglib prefix="render" uri="futuretense_cs/
    render.tld"%>
4   <%@ taglib prefix="string" uri="futuretense_cs/
    string.tld"%>
5   <%@ taglib prefix="asset" uri="futuretense_cs/asset.tld"%>
6   <%@ taglib prefix="insite" uri="futuretense_cs/
    insite.tld"%>
7   <cs:ftcs>
8   <div id="StandardDetailView">
9   <ics:if condition='<%=ics.GetVar("tid")!=null%>'><ics:then>
    <render:logdep cid='<%=ics.GetVar("tid")%>' c="Template"/>
    </ics:then></ics:if>
10
11  <%-- load the page, display the title --%>
12
13  <asset:load name="page" type="Page"
    objectid='<%=ics.GetVar("cid")%>'/>
14  <asset:get name="page" field="description"
    output="description"/>
15  <h2><string:stream variable="description"/></h2>
16
17  <%-- create the first slot --%>
18  <insite:calltemplate
19              slotname="Slot01"
20              site='<%=ics.GetVar("site")%>'
21              tid='<%=ics.GetVar("tid")%>'>
22      <insite:argument name="p" value='<%=ics.GetVar("p")%>'/>
23      <insite:argument name="locale"
    value='<%=ics.GetVar("locale")%>'/>
24  </insite:calltemplate>
25
26  <%-- create the second slot --%>
27  <insite:calltemplate
28              slotname="Slot02"
29              site='<%=ics.GetVar("site")%>'
30              tid='<%=ics.GetVar("tid")%>'>
31      <insite:argument name="p" value='<%=ics.GetVar("p")%>'/>
32      <insite:argument name="locale"
    value='<%=ics.GetVar("locale")%>'/>
33  </insite:calltemplate>
34
35  </div>
36  </cs:ftcs>
```

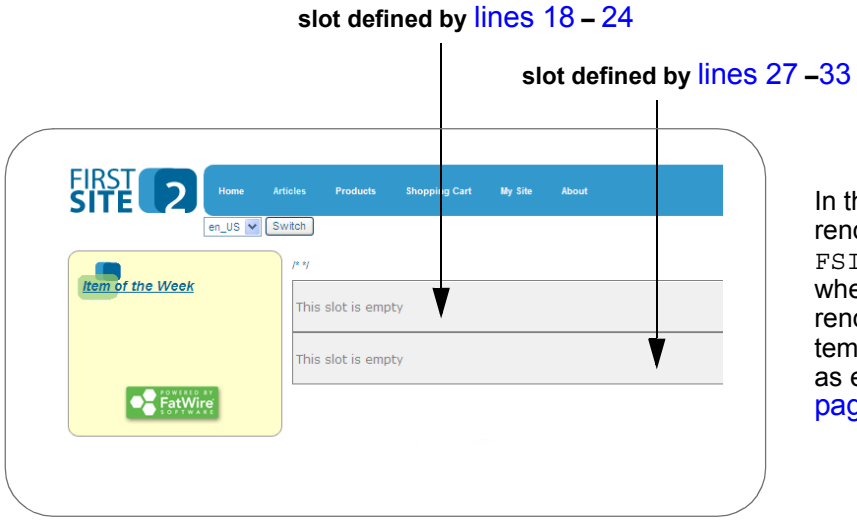
**Note**

The sample code above was used in the FirstSite II sample site to replace the element logic of the `FSIIStandardDetailView` Page template. Because this template is called by `FSIILayout` to render the main area of the "Home," "Articles," and "About" pages, the main area now displays two slots. The remaining areas are rendered by `FSIILayout`.

2. To use (and test) the master page:
   a. The user previews an asset that is rendered by the master template (either directly or via a caller template). At this point, the user is operating within the InSite interface.

      (In this example, the user previews an asset of type "Content" in the "Articles" page on the FirstSiteII sample site.)

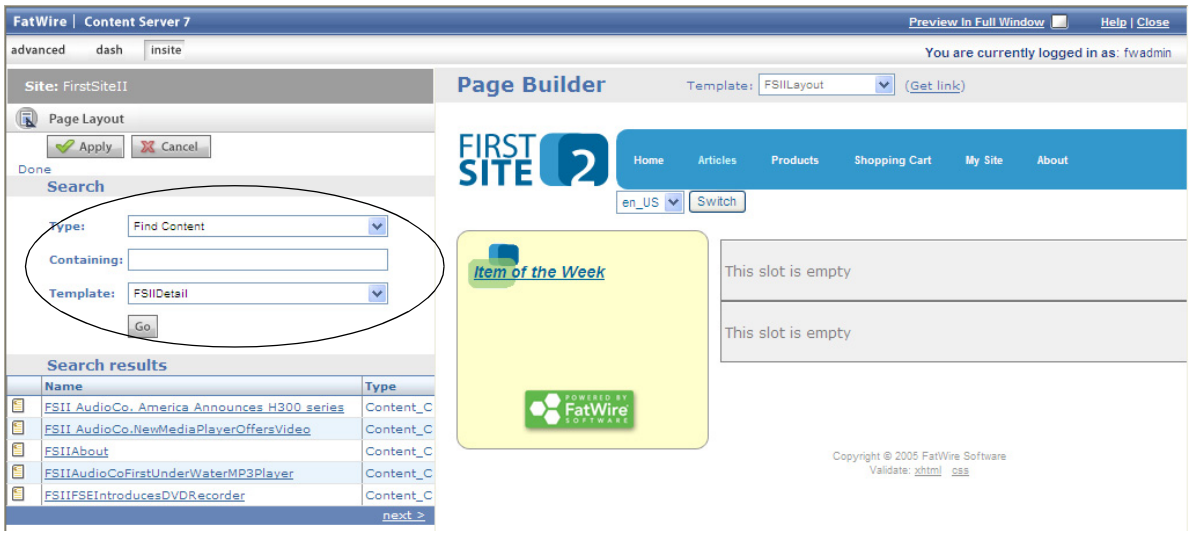
   b. The user clicks the **Page Layout** bar to display the master page (similar to the page in the figure below).



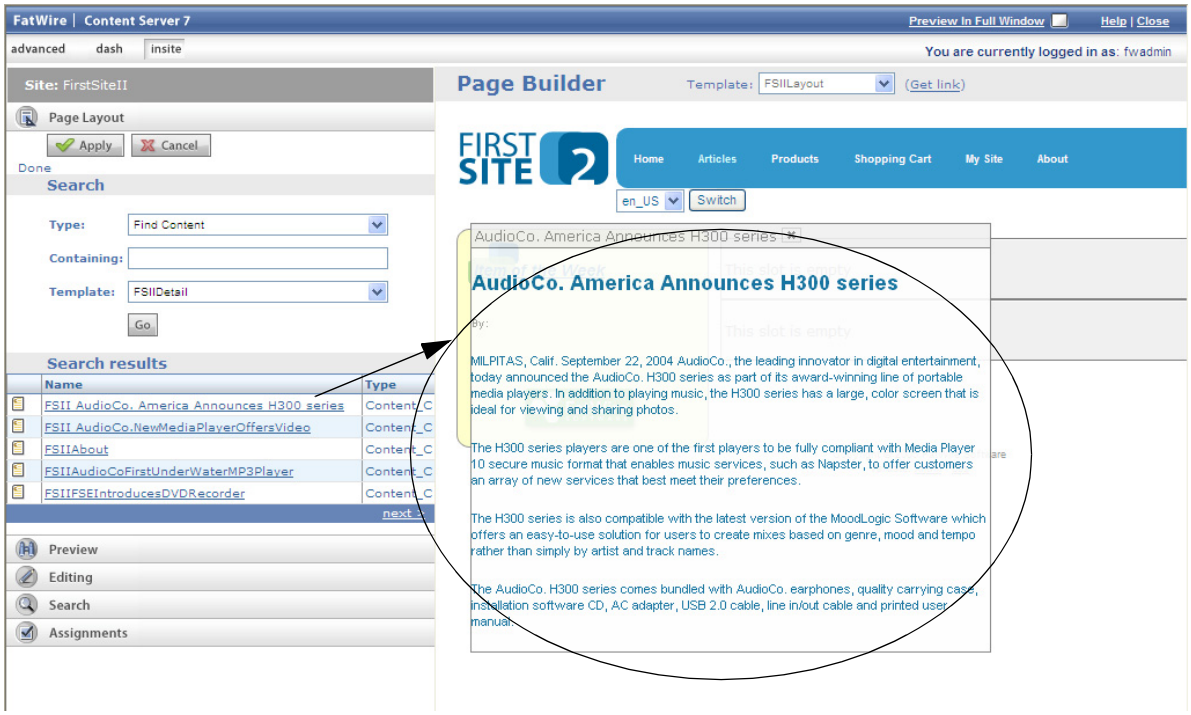

In this example, the slots are rendered by the template `FSIIStandardDetailView`, whereas the rest of the page is rendered by the caller template `FSIILayout`, as explained in the note on page 546.

3. To select content for a slot:

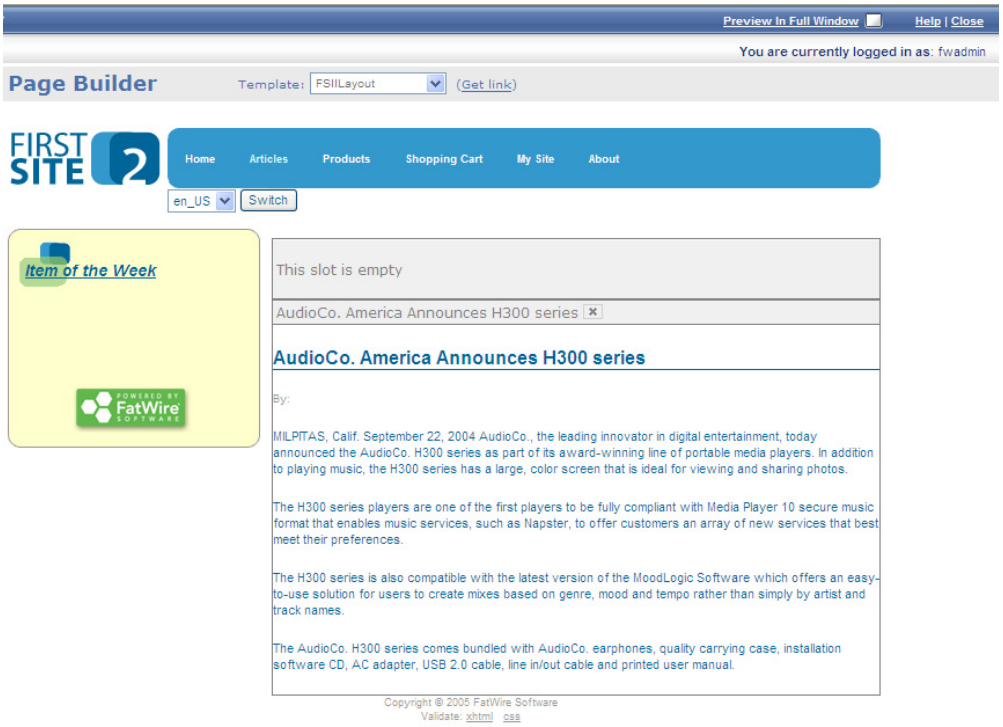
**a.** The user sets up a search for asset-template pairs (that is, the user searches for assets of a given type and names the template that will render the assets in the desired format).




**b.** From the search results list, the user selects an asset. Using the named template, Content Server renders the asset as a floating object.



4. The user drags the object into a slot, creating the page shown below. The master template is automatically edited to reflect the change to the slot.




# Template Context

"Context" refers to the reusability of a master template, meaning that a master template can be used in many places on a site at any one time; any given slot can hold a different asset-template pair, depending on the page (i.e., context) in which the slot is used. Starting with Content Server 7.0, context is active by default.

For example, consider the `FSIIStandardDetailView` template, which contains the code sample on page 545. When three pages ("Home," "Articles," and "About") are first previewed in the Page Layout utility, the main part of each page initially displays two empty slots (slot 01 and slot 02) as shown on page 546. On all the pages, the slots are rendered by the same `FSIIStandardDetailView` template. On the individual pages, the same slot (for example, slot02) can be populated with a different asset-template pair, each pair unique to its page (context).

Slots can be made context-independent, by overriding the `context` attribute in the `<insite:calltemplate>` tag with a constant value (for example, `context="all"`, or `context=" "`). In our example, slot02 would then display the same content, regardless of the page (context). For the user of the master template, this means having to drag and drop an asset-template pair only once into a given slot.

# Guidelines for Creating Master Templates

A master template must be either a JSP or XML template. It is like any other template that can be created in Content Server, except for the slots it contains.

To create a master template, follow instructions in Chapter 23, "Creating Template, CSElement, and SiteEntry Assets" and adjust your procedure, using the guidelines below:

1. In the "Name" screen of the Template form:
   a. Name the template in a way that will identify it as an InSite master template, so that users will be able to distinguish it from templates of other types.
   b. Make sure the field "For Asset Type" correctly identifies the template as typed or typeless. (Slots can be created in both typed and typeless JSP templates.)
      - If you are creating a typeless template, select **can apply to various asset types**.
      - If you are creating a typed template, select the asset type you require.
2. In the "Element" screen of the Template form:
   a. Set the **Usage** field to either **Usage unspecified** or **Element defines a whole HTML page**.
   b. In the **Create Template Element** field, select **JSP**.
3. When coding the element logic (in the "Element" screen of the Template form):
   - Insert the following line into the header:

     ```
     <%@ taglib prefix="insite" uri="futuretense_cs/insite.tld"%>
     ```

     (This line imports the library that enables you to use the `<insite:calltemplate>` tag.)
   - Create slots in the template by using the following tag for each slot:

     ```
     <insite:calltemplate
        site="site name"
        slotname="name of slot" (the name must be unique)
        tid="caller template or CSElement id"
        [ttype="Template or CSElement"]
        [c="asset type"]
        [cid="asset id"]
        [tname="target template or CSElement name"]
        [context="context override"]
        [style="pagelet or element"]
        [packedargs="packed arguments"]
       <insite:argument name="" value=""/>
       <insite:argument name="" value=""/>
     </insite:calltemplate>
     ```

**Note**

- Slots must be named uniquely within the template code (otherwise, the master template cannot be saved).
- Slots can be nested.
- Slots can be formatted. For example, slots in the context of a table can be formatted to display borders of a specific width and color. (Note also that best practices advises against using tables to specify page layout, as their function is to tabulate data.)
- When the user graphically makes changes to the content of a slot, the template code is automatically updated to reflect the changes.

For an example of how the `<insite:calltemplate>` tag is used in context, see page 545. For more information about the `<insite:calltemplate>` tag, see the *Content Server Developer's Tag Reference*.

4. When you finish creating the master template, test it by using it in the same way that a content provider would. For instructions, see "Managing Page Content Using the InSite Interface," in the Content Server user's guides.

   When testing the template, bear in mind that changes to a slot result in automatic changes to the underlying template code.

5. If you need instructions on sharing or otherwise managing the template, see "Managing Template, CSElement, and SiteEntry Assets," on page 537 in this guide.

# Tracking Changes to Master Pages

When a master page is saved, Content Server records the final changes (if any) to the slots in the `Template_Composition` table (one row per slot). For example, an initially empty slot in a master template named "`IST_Template`" is filled with an asset of type `Content` (rendered by a template named "`Detail`"). When the master template is saved, the following information about the slot is recorded in the `Template_Composition` table.

| Data | Description |
|---|---|
| `id` | Arbitrary number that is assigned to a row of data to identify the data. |
| `cs_ownerid` | ID of the master template (which holds the slot). In this example, the name is `IST_Template`. |
| `cs_slotname` | Name of the slot. |
| `cs_site` | Site that was affected by the change. |
| `cs_assettype` | Type of asset that fills the slot (the `c` parameter. `Content`, in this example). |
| `cs_assetid` | ID of the asset that fills the slot (the `cid` parameter). |

| Data | Description |
|---|---|
| `cs_tname` | Name of the template that renders the asset (`Detail` template, in this example). The names of typeless templates begin with a forward slash ( / ).<br>Note that *`cs_assetid`* and *`cs_tname`* form an asset-template pair, which defines the rendered asset. |

Chapter 25

# Creating Collection, Query, Stylesheet, and Page Assets

The core asset types delivered with CS-Direct provide basic site design logic. This chapter describes how to create the page, query, and collection assets that implement the functionality of your online site. It also includes a section for the stylesheet asset type, a sample asset type delivered with the Burlington Financial sample site.

The preceding chapter describes how to create Template assets. Because you assign Template assets to your other assets, it is typical to create your templates before you create your site design asset types.

The procedures for working with assets of any type are very similar and are described thoroughly in the Content Server user's guides. This chapter presents procedures that are unique for the collection, query, stylesheet, and page asset types. It contains the following sections:


- Previewing Assets
- Approving Assets
- Sharing Assets
- Deleting Assets
- Collection Assets
- Query Assets
- Stylesheet Assets
- Page Assets

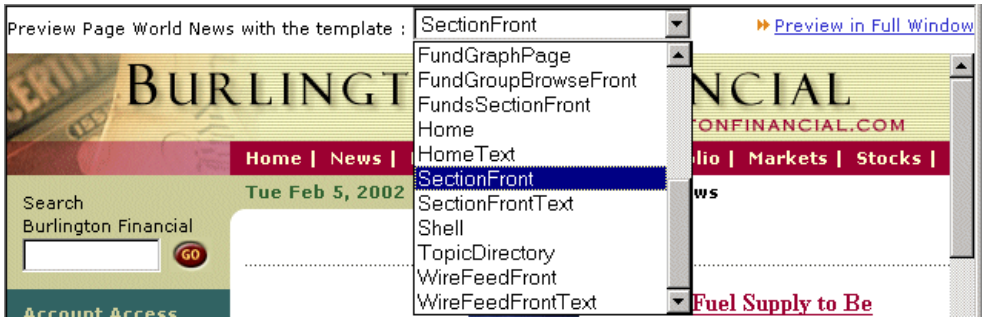
# Previewing Assets

You can preview any asset that has a Template asset selected in its **Template** field. The preview feature also lets you select other templates to use to preview the same asset.

### To preview an asset

- Right-click on the asset in the tree view on the **Site Plan** tab, and select **Preview** from the pop-up menu.
- Inspect or edit the asset in the work area, and click the **Preview** icon.
- Search for a list of assets, and click the **Preview** button next to an asset in the list of search results.

CS-Direct renders the asset and displays it in a new browser window, using the template you have assigned as the default format. To preview the asset with a template other than its default template, select the template that you want to use from the drop-down list of templates displayed in the browser, as shown here:




**Note**

Depending on how your templates are written, some of them may not display your asset correctly in the context of the management system.

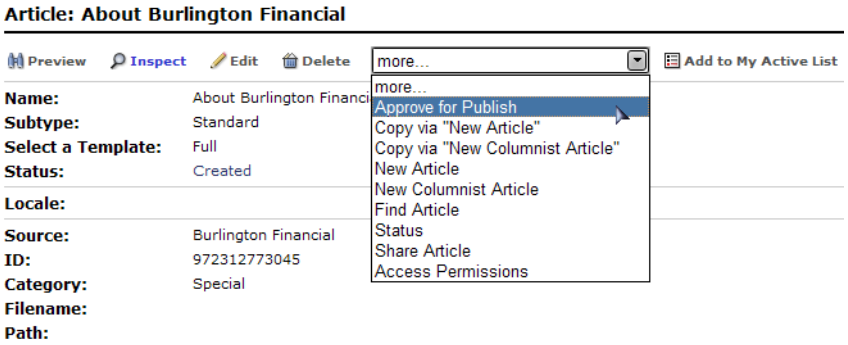
# Approving Assets

The publishing process that either copies assets from one system to another system (Mirror to Server) or renders assets into static files (Export to Disk) is a background process that is typically configured to run at regularly scheduled times. The publishing process publishes only those assets that have been **approved**.

When you approve an asset, the approval system examines the asset to determine if it has any dependencies on other assets. For example, if the assets in an approved collection are not approved, the collection cannot be published.

Depending on how your development and management systems are set up, approving assets for publish might be a workflow step—typically the last workflow step in a workflow process. For more information about approvals and publishing, see the *Content Server Administrator's Guide*.

To approve an asset for publish, you select **Approve for Publish** from the drop-down list in the “Inspect” or “Status” form for the asset, as shown here:




If there is more than one publishing destination set up, you then must select which publishing destination the asset is approved for.

# Sharing Assets

If you want to use your assets in more than one site, you can share them so you do not have to create and maintain the same asset more than once.

Before you share an asset, consider the following tips:

- You can share an asset only to sites that you have access to. If you have access to only one site, the **Share Assets** function is not available to you.
- You cannot share page assets.
- Share an asset only if it must be identical on all sites it is shared with; that is, do not share an asset if you need to make any modifications for one of the sites it is shared with. In that case, create a new asset for the site that needs the modifications.
- Be sure that the Template assets assigned to shared assets are appropriate. The Template assets themselves must be shared. Otherwise, you will be unable to preview the asset on the site you share it with.
- If the shared asset has a workflow assigned to it, you and others can change its workflow status only when you are working in the asset’s original site.
- It is good practice to share the asset only when you are ready to publish it; that is, to not share the asset until it has been approved. There is a workflow privilege called **Share Assets,** which means that your site administrator can set up a workflow that enforces this practice.

For the basic procedure for sharing assets, see the *Content Server User’s Guide*.

# Deleting Assets

The **Delete** function does not actually remove an asset from the database. A better description is that it marks it as deleted.

When you delete an asset, CS-Direct does the following:

- Changes the value of the asset's Name column in its main storage table to its object ID.
- Changes the value of the asset's Status column to VO, for "void."
- Approves the asset for publishing to any destination it has ever been published to.

The following restrictions are enforced when you delete assets:

- You can delete an asset only if you have the privileges to do so.
- Even if you have the ability to delete assets, you cannot delete an asset that is assigned to someone other than you if you are using the workflow feature.
- You cannot delete an asset if it has associations with other assets. For example, you cannot delete an article if it is included in a collection. You must remove the article from the collection before you can delete the article. (CS-Direct displays information about an asset's associations when this situation occurs.)

For the basic procedure for deleting assets, see the Content Server user's guides.

# Collection Assets

A collection asset stores an ordered list of assets of one type. You **create** (or design) a collection asset by naming it and selecting query assets for it. By default, you can select up to three query assets. If your site design requires more queries for collection assets, you can create additional named associations for the additional queries. For information about creating associations, see "Step 9: (Optional) Configure Association Fields" on page 296.

A collection uses a query asset to obtain a list of possible assets for the collection. You **build** (or populate) a collection by running its queries, selecting assets from the results of the queries, and then ranking and ordering the assets that you selected. This ranked, ordered list is the collection.

Using collections is one way to keep the content displayed on rendered pages current and up-to-date. The Burlington Financial sample site uses several collections. For example, you can select a collection in the Top Stories field for a Burlington Financial page asset. A publisher or content provider can then change the content identified by that association by doing one of the following:

- Selecting a different collection from the tree
- Building the assigned collection and selecting different assets in it

## Before You Begin

Before you **create** collection assets, note the following:

- A collection must have at least one query, so be sure that you create the queries before you try to create your collections.

- Because you assign templates to collections, you should also create the Template assets before you create your collection assets.

Before you **build** the collection, you should determine how the Template asset assigned to it is coded. For example, if you select 100 assets for a collection but the template is coded to display only five of them, the following occurs:

- The rendered page that displays those assets displays only the first five.
- The page takes longer to render than necessary because CS-Direct has to sort through all 100 assets even though it displays only the first five.

For more information about building a collection, see the *Content Server Advanced Interface User's Guide*.

## Creating Collection Assets

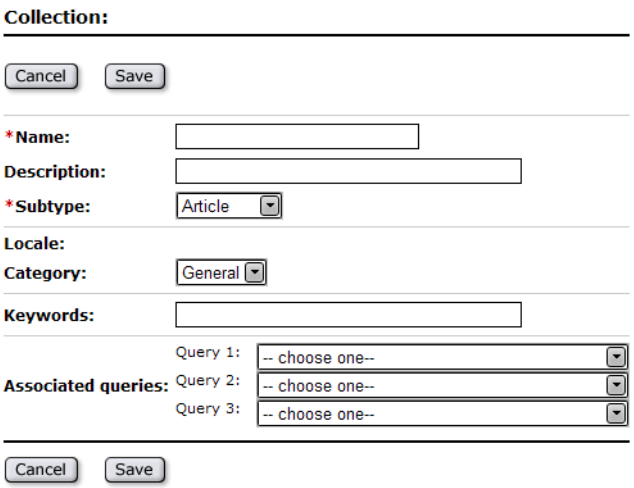
### To create a collection asset

**Note**

To use this procedure, you must have Collection asset types enabled for the site you are working in. Step 4 indicates whether they are enabled.

1. If necessary, log in to the Content Server interface.
2. Make sure that you have completed the steps in section "Before You Begin" on page 556.
3. Click **New** on the button bar.
4. Select **New Collection** from the list of asset types. (If Collection asset types are not enabled, the option is not displayed.)

   The "Collection" form appears:




5. (Required) In the **Name** field, type a descriptive name for the page. You can enter up to 64 alphanumeric characters, but the first character must be a letter. Underscores (_) and hyphens (-) are acceptable, but tab and space characters are not.

6. In the **Description** field, type a brief description of the collection. You can enter up to 128 characters.
7. (Required) In the **Subtype** field, select the type of asset this collection will hold.

> **Note**
>
> Collection subtypes are controlled by Query. When a query is set up for a certain asset type, that asset type becomes a value of the **Subtype** field. The **SubType** field thus lists every asset type for which a query was created. For information about creating Query assets, see page "Creating Query Assets" on page 562.)

8. In the **Select a Template** field, select a Template asset from the drop-down list.
9. In the **Category** field, select a category from the drop-down list. (If you do not select a category, the first item on the list is selected by default.)
10. In the **Keywords** field, enter keywords that you and others can use as search criteria in the "Advanced Search" form when you search for this collection in the future. For information about searching for assets, see the *Content Server Advanced Interface User's Guide*.
11. In the **Associated queries** section, select up to three queries. All of the queries that you select for this collection must return assets of the same type.
12. Click **Save**.

## Sharing Collection Assets

Before you share a collection asset, consider the following:

- Building a collection in one site builds it in all of the sites that it is shared with. You cannot build a collection to include different assets for different sites.
- The query assets used in the shared collection must be coded to return only assets that are shared to all the sites that the collection is shared with.
- As with any shared asset, be sure that the template assigned to the collection is also shared to the other site.

For the basic procedure for sharing assets, see the *Content Server Advanced Interface User's Guide*.

# Query Assets

A query asset stores a database query that retrieves a list of other assets from the database. However, if the query is to be used for a collection, it can return assets of one type only.

## Query Assets and Other Assets

CS-Direct uses queries differently in collection assets than it does for other assets:

- When you build (or populate) a collection, you run one or more query assets and then select and order the assets that you want from the resulting list. The collection is a **static** list of assets selected from the query resultsets.
- You can select queries for a page asset either through informal relationships or through named associations. You can select queries for other asset types (article, for example) through named associations.

  When the asset is rendered, it does not invoke the query directly. Either the template element that formats the asset or a template element that formats the query is coded to invoke a standard CS-Direct element called:

  `OpenMarket/Xcelerate/AssetType/Query/ExecuteQuery`

  This element runs the query asset when the asset it is associated with is rendered, which means the resultset is **dynamic**.

## How the Query Is Stored

A query asset can store its database query in one of two ways:

- Directly. You can write the query directly into the **SQL query** field of the "Query" form. You can either use standard SQL for the query, or, if your CS-Direct systems use an external search engine, you can use an appropriate search engine query.
- Indirectly. You can write the query in an element and then store the location of that element in the query asset by identifying it in the **Element name** field in the "Query" form. An element for a query is like any other element: you can use XML, JSP, JavaScript, HTML, and so on.

Most of the Burlington Financial queries store the query directly; that is, the SQL query is written directly into the **SQL query** field in the "Query" form. For example, the following code is from the News Wire Feed Query:

```
SELECT DISTINCT Article.id, Article.name, Article.updateddate,
Article.subheadline, Article.abstract, Article.description,
Category.description AS category, StatusCode.description AS
statusdesc FROM Article, Category, AssetPublication, StatusCode
WHERE Article.status!='VO' AND Article.category=Category.category
AND Article.status=StatusCode.statuscode AND
Category.assettype='Article' AND Article.source='WireFeed' AND
Article.category='n' AND Article.id = AssetPublication.assetid AND
AssetPublication.pubid = 968251170475 ORDER BY Article.updateddate
DESC
```

## Commonly Used Fields for Queries

There are several CS-Direct fields, four of which are used in the preceding News Wire Feed query example, that you are likely to use in your queries:

- status
- updateddate
- source
- category
- pubid

- startdate
- enddate

The rest of this section defines the fields in this list.

## status

All assets have a **status**. When an asset is created, CS-Direct adds a row to the table that holds assets of that type and sets its status to PL, which means "created."

The following table lists and defines the status codes that CS-Direct uses:

| Status Code | Definition |
|---|---|
| PL | created |
| ED | edited |
| RF | received (from XMLPost, for example) |
| UP | upgraded from Xcelerate 2.2 |
| VO | deleted (void) |

These codes are listed in the `StatusCode` table in the database.

When an asset is deleted, CS-Direct changes its status to VO and renames the string in its Name field to its object ID.

Write your queries to exclude assets whose status is VO. For example: `WHERE Article.status!='VO'`

## updateddate

The information in the **updateddate** field represents the date on which the information in the status field was changed to its current state. Depending on the design of your site, you might want a query to return assets based on this date.

## source

The **source** field is a default CS-Direct field that can identify where an asset originated. It is not required.

For example, the Burlington Financial sample site has sources named WireFeed, Asia Pulse, UPI, and so on. (You add sources for your sites on the **Admin** tab in the tree. See "Step 11: (Optional) Configure Sources" on page 299.)

If you use source with your assets, you can write your queries to use source as a parameter. In the previously mentioned News Wire Feed query example, the `AND Article.source='WireFeed'` statement ensures that only articles with WireFeed in their Source fields are selected by this query.

## category

The **category** is a default CS-Direct field that can categorize assets according to a convention that works for your sites. It is not required.

For example, the Burlington Financial sample site has categories named Personal Finance, Banking and Loans, Rates and Bonds, News, and so on. (You add categories for your sites on the Admin tab in the tree. See "Step 10: (Optional) Configure Categories" on page 298.)

If you use category with your assets, you can write your queries to use category as a parameter. In the previously mentioned News Wire Feed query example, the `Article.category='n'` statement includes article assets from the News category.

### pubid

A **pubid** is a unique value that identifies a site (or, in old terminology, a publication). When an asset is created, CS-Direct writes information about that asset to several database tables, one of which is the `AssetPublication` table.

An asset's row in the `AssetPublication` table includes the pubid of the site the asset was created for. If the asset is shared, the `AssetPublication` table has a row for each site that the asset is shared with. For example, if an article asset is available in two sites, there are two rows for that article in the `AssetPublication` table.

If you have only one CS-Direct site on your system or if your query results do not need to be site-specific, you do not need to code your queries to consider pubid. The Burlington Financial queries, however, are coded to restrict assets based on the pubid of Burlington Financial (`AssetPublication.pubid = 968251170475`) so that they do not return assets from another site.

### startdate and enddate

Neither of the sample sites use the **startdate** and **enddate** fields but the CS-Direct database has columns to store this information. These fields exist so that you can assign time limits to assets.

If your asset types use the startdate and enddate fields, you can create queries that select assets based on the dates stored in those fields.

## Before You Begin

Before you begin creating query assets, consider the following:

- Query assets that are used on assets other than collections are not required to have templates. You can either create template elements specifically for your query assets that identify, run, and display the results, or you can code the template elements for your page assets to do that.
- When you write a query for a collection, be sure to code it to select the fields that are required for that asset type. CS-Direct is programmed to expect information from an asset type's required fields so that it can display that information in the "Build Collection" form.

  For example, the **Name** and **Description** fields are required fields for a Burlington Financial article. (The **Description** field is renamed and displayed as **Headline** in the form.) Therefore, the queries for Burlington Financial collections that hold Burlington Financial articles select the **Name** and **Description** fields. Those queries also select several other fields, but CS-Direct requires at least the **Name** and **Description** fields to present the assets returned by the queries in the **Build Collection** forms correctly.

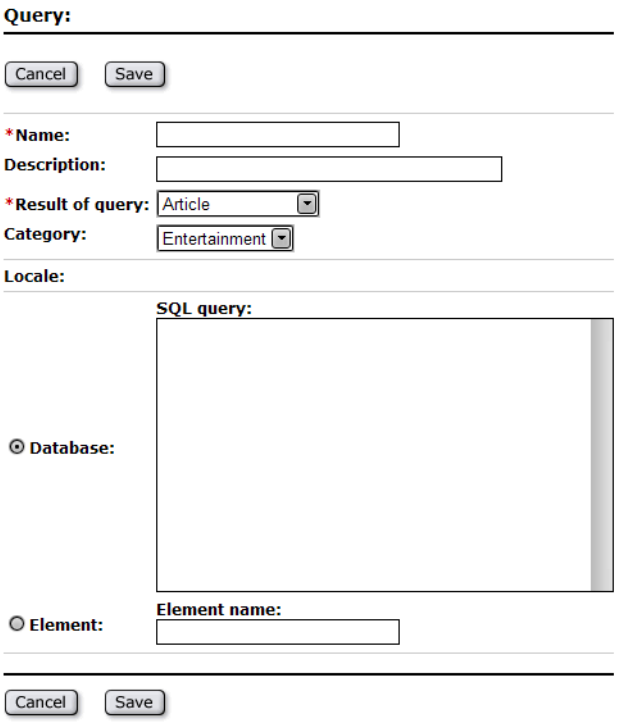
- Query assets that are used only for collections have no need for templates. The template element assigned to the collection formats the assets in a collection's list of assets.
- For performance reasons, be sure to create efficient queries. For example:
  - Include as much logic as possible in the query rather than in the element that runs and displays the results of the query. For example, if you want to filter or constrain a list of articles, be sure the query performs the filtering or constraining step so that the list returned to the element is complete rather than coding the query to return the entire list and using the element code to constrain the list.
  - Be sure your queries return only the information that the element displays.
- Query assets that are for collections must return assets of one type only.

## Creating Query Assets

### To create a query asset

1. If necessary, log in to the Content Server interface.
2. Click **New** on the button bar.
3. Select **Query** from the list of asset types. (Query asset types must be enabled for your site.)

   The "Query" form appears:




4. (Required) In the **Name** field, type a descriptive name for the query asset. You can enter up to 64 alphanumeric characters, but the first character must be a letter. Underscores (_) and hyphens (-) are acceptable, but tab and space characters are not.

5. In the **Description** field, type a brief description of the query. You can enter up to 128 characters.
6. In the **Template** field, select a Template asset from the drop-down list.
7. In the **Category** field, select a category from the drop-down list. (If you do not select a category, the first item on the list is selected by default.)
8. In the **Result of query** field, select the type of asset that this query returns. (The query can return assets of one type only if this asset is to be used by a collection.)
9. Do one of the following:
   - If you want to store the query directly in this asset, select **Database**, click in the **SQL query** field, and then write your query.
   - If you wrote the query in an element, select **Element** and then enter the entire name of the element in the **Element name** field.
10. Click **Save**.

## Sharing Query Assets

If you plan to share a query asset with another site, consider the following tips:

- If you want your query results to be site-specific, be sure to include a `WHERE` clause for `pubid` so that the query does not return assets to a site where those assets have not been shared.
  - For example, in either a query for a collection or a query for a static site, you can use the following statement:

    ```
    WHERE AssetPublication.pubid = SessionVariables.pubid
    ```

    because `SessionVariables.pubid` is always set when you are building a collection or using the Export to Disk function.
  - If the query is to be used on a dynamic site, you can use that same statement as long as you code your elements to either pass in the identify of `pubid` to the `ExecuteQuery` element or to set the `SessionVariables.pubid` variable.
- Because page assets cannot be shared, you should not share query assets if they return page assets.
- As with any shared asset, if the query has a template, be sure that the template assigned to the query is also shared with the other site.

For the basic procedure for sharing assets, see the *Content Server Advanced Interface User's Guide*.

## Previewing and Approving Query Assets

First, remember that not all query assets have their own templates. If a query asset was designed to be used on a page asset and it is the page asset's template that actually formats the query, you must preview the page in order to preview the query.

If your online site is a dynamic site — that is, you use the Mirror to Server publishing method—a query asset might return different assets on the management system than it does on the delivery system, depending on which assets have been published.

Therefore, if you preview your query to determine whether you should approve it or not, remember that the assets that it returns on the management system (where you are

previewing it) could be different than the assets that it will return on the delivery system after it is published.

# Stylesheet Assets

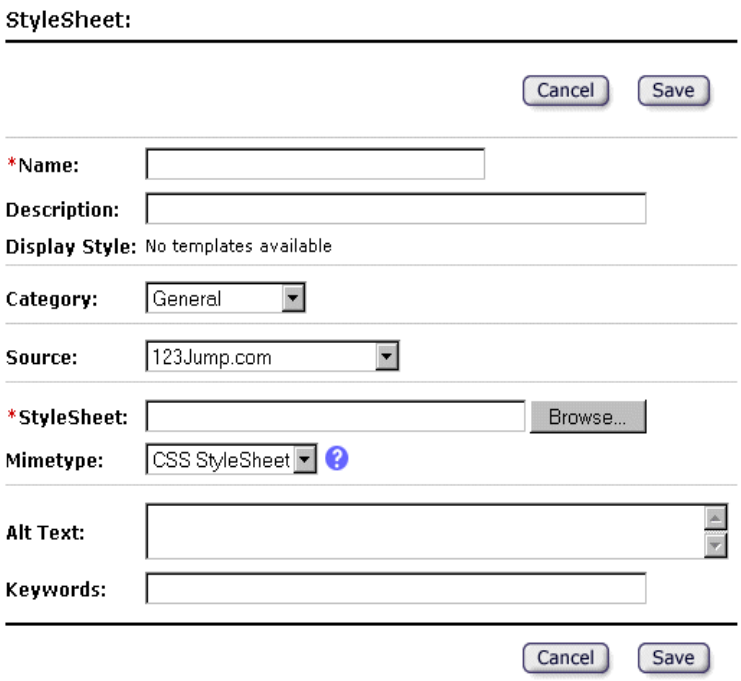
Stylesheet assets store style sheet files of any format (CSS, XSL, and so on). This asset type is installed only if you install the Burlington Financial sample site. Typically, you do not assign Template assets to stylesheet assets because they are, effectively, templates in themselves. However, if you need to create a different kind of stylesheet or you want to display information about the stylesheet on a site, you can certainly create a Template asset and assign it to a stylesheet asset.

## Creating Stylesheet Assets

### To create a stylesheet asset

1. If the Content Server interface is not already open, log in.
2. Click **New** on the button bar.
3. Select **Stylesheet** from the list of asset types.

   The new “Stylesheet” form appears:




4. (Required) In the **Name** field, type a unique, descriptive name for the stylesheet. You can enter up to 64 alphanumeric characters, but the first character must be a letter. Underscores (_) and hyphens (-) are acceptable, but tab and space characters are not.
5. In the **Description** field, type a brief description of the stylesheet. You can enter up to 128 characters.

6. (Optional) In the **Template** field, select one from the drop-down list.
7. In the **Category** field, select a category from the drop-down list. (If you do not select a category, the first item on the list is selected by default.)
8. In the **Source** field, select one from the drop-down list.
9. Do one of the following:
   - In the **Stylesheet** field, enter the full path and file name of the stylesheet.

     **Note**

     Some browsers do not allow you to type into the **Stylesheet** field.

   - Click **Browse** next to the **Stylesheet** field and select the stylesheet file.
10. In the **Mimetype** field, select the mimetype of the file you specified in the previous step from the drop-down list. If the correct mimetype is not displayed in the list, see "Step 12: (Conditional) Add Mimetypes" on page 300.
11. In the **Keywords** field, enter keywords that you and others can use as search criteria in the "Advanced Search" form when you search for this template in the future. For information about searching for assets, see the *Content Server Advanced Interface User's Guide*.
12. Click **Save**.

    CS-Direct uploads the file you specified in the **Stylesheet** field.

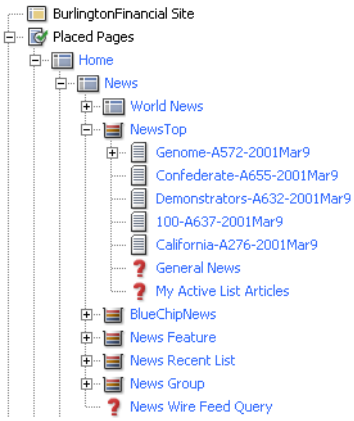
## Sharing Stylesheet Assets

Stylesheet assets are standalone which means that you can share them without considering dependencies with other assets.

# Page Assets

Page assets are site design assets that store references to other assets, organizing them according to the design that you and the other designers are implementing.

Open CS-Direct and examine the page assets listed in the site tree for the Burlington Financial sample site:




These page assets represent sections of the site, in essence the structure or organization of the site. They do not represent each and every rendered page that can possibly be served. This structural organization is primarily for the benefit of your CS-Direct users. This is not the only way of organizing your site, but it is convenient for your editors to see a structure that resembles your finished website.

Typically, you create page assets once: when you design the site. You associate collections, queries, articles, and so on with page assets and you code template elements that format the types of assets you want to associate with the page asset.

Before you can select the correct content for your page assets, you must be familiar with how your site is structured and what your template elements for page assets are designed to do. That is why you and other site developers—the people who are coding elements and creating Template assets—typically create the page assets for a site.

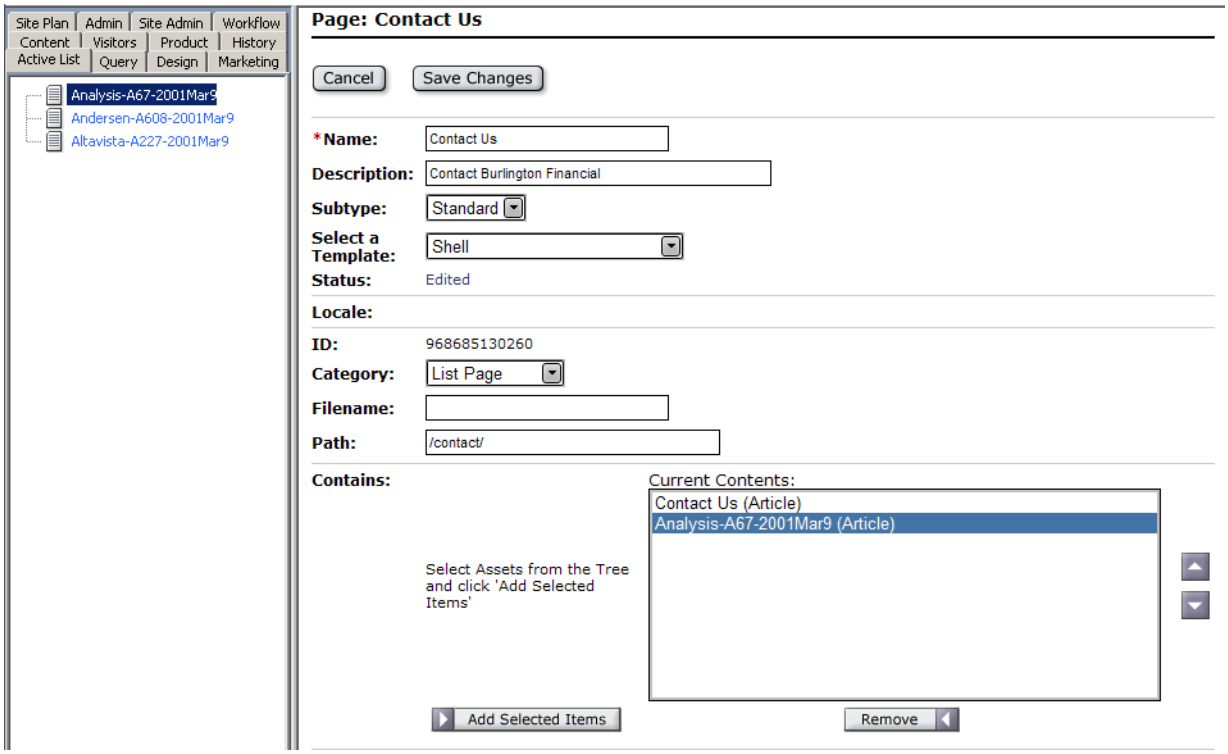
## Creating a Page Asset

1. If necessary, log in to CS-Direct, and if given a choice, select a site.
2. Click **Search** on the button bar and run as many searches as necessary to find all the articles, queries, images, collections, or other assets that you want to include on the page.

3. In the search results list, select the check box on the right, next to the name of each asset that you want to display on the new page, and click the **Add to My Active List** button at the bottom of the list, as shown here:

4. CS-Direct displays an updated My Active List page on the right, and also lists the assets on the **My Active List** tab in the tree view on the left:
5. Click **New** on the button bar.
6. Select **Page** from the list of asset types. (Page asset types must be enabled for your site.) The “Page” form appears.
7. (Required) In the **Name** field, type a descriptive name for the page. You can enter up to 64 alphanumeric characters, but the first character must be a letter. Underscores (_) and hyphens (-) are acceptable, but tab and space characters are not.
8. In the **Description** field, type a brief description of the page. You can enter up to 128 characters.
9. In the **Template** field, select a Template asset from the drop-down list.
10. In the **Category** field, select a category from the drop-down list. (If you do not select a category, the first item on the list is selected by default.)
11. If you are creating page assets for exporting to a static site and you are using the **Filename** and **Path** fields, enter the appropriate file name information according to the conventions that your organization is using. For information about how the Export to Disk publishing process uses this information, see the *Content Server Administrator’s Guide*.

12. To add items from your active list to the **Current Contents** list on the page form, select and highlight items on the **Active List** tab in the tree view, and click **Add Selected Items**, as shown here:




Whether you select assets from the **Current Contents** section, the **Related** section, or some combination of both to appear on the page asset depends on how you have coded the template element for the page.

Selecting assets from the **Contains** section creates unnamed relationships between the asset and this page asset. The fields in the **Related** section represent named associations. In both cases, the asset becomes a child asset of this page asset, and you can then use an `ASSET.CHILDREN` tag to return those assets.

For information about the `ASSET.CHILDREN` tag, see the *Content Server Tag Reference*. For information about named associations and unnamed relationships, see "Relationships Between Basic Assets" on page 187.

Adding items to the **Contains** section does not guarantee that those assets will actually appear on the rendered version of the page asset. The template element that you select from the **Template** field must be coded to identify and display the types of assets that are in the **Contains** section, otherwise those assets cannot be rendered.

This also applies to the assets that you select in the **Related** section of the form. Selecting assets from fields that designate named associations between the page asset and assets of other types does not guarantee that those assets will appear on the rendered version of the page asset. The template element must be coded to display them, otherwise, those assets cannot be rendered.

13. Use the arrows on the right side of the **Contains** list to position the assets in the correct order. This determines the order of the assets on a rendered page.

14. In the **Related** section of the form, select the assets that you want to use as the named associations for the page.

    The assets that you select in this section become child assets of this page asset.

    **Note**

    See the *Content Server Advanced Interface User's Guide* for instructions concerning collection assets.

15. Click **Save**.

    The page is saved. It now appears on the site tree under the **Unplaced Pages** page. To place the page, see "Placing Page Assets" below.

## Placing Page Assets

After you create a page asset, you position it in the appropriate location in the site tree by using the **Place** function.

### To place a page asset

1. If necessary, log in to the Content Server interface.
2. Click the **Site Plan** tab, where you should see the site tree with the new page asset in the **Unplaced Pages** list, as shown here:
3. Expand the list of **Placed Pages** in the site tree.
4. Select a parent for the page you are placing by doing one of the following:
   - If you want to place a page at the top-most level in the tree, right-click on the **Placed Pages** icon.
   - Otherwise, right-click on the placed page under which you want to insert the new unplaced page, and choose **Place Page** from the pop-up menu.

The place page form appears in the work area on the right. It lists all child pages that are placed under the parent page. It also lists all pages that have not yet been placed in the site tree:

5. Place the page:
   a. In the list of unplaced pages, type a number in the **Rank** field to designate the new page's position in the list of child page assets. Position numbering starts at 1, the top of the list. (In this example, it makes sense to type "1".)
   b. Click **Save**.

      The unplaced page asset moves to the site tree, to its assigned rank. (To view the page asset in its new location, you may need to right-click in the site tree and choose **Refresh All** from the pop-up menu.)
6. Preview both the parent page and the placed child page. (See "Collection Assets" on page 556 for instructions.)

# Moving Page Assets in the Site Tree

In addition to placing unplaced pages, you can also use the place page form to:

- Change the order of child pages within the same parent page.
- Move a child page from one parent page to another.

## Re-ordering Child Pages

### To re-order children of the same parent page

1. If necessary, log in to the Content Server interface.
2. Click the **Site Plan** tab and expand the list of **Placed Pages** in the site tree.

3. Right-click on a placed page that has more than one child page, and choose **Place Page** from the pop-up menu.

   The place page form appears in the work area on the right.
4. In the list of placed child pages, type new values in the **Rank** column to re-order the child pages.
5. Click **Save**.

   The child pages move to their new positions in the site tree.

## Changing Parent Pages

### To move a child page from one parent page to another

1. If necessary, log in to the Content Server interface.
2. Click the **Site Plan** tab and expand the list of **Placed Pages** in the site tree.
3. Remove the page asset from its parent page:
   a. Right-click on the placed page whose child page you want to move, and choose **Place Page** from the pop-up menu.

      The place page form appears in the work area on the right.
   b. In the list of placed child pages, select the **Remove** check box next to the child page that you want to move, as shown here:

      **Pages that have been placed under FirstSiteII:**

      | Rank | Remove? | Name | Description | Modified |
      |---|---|---|---|---|
      | 1 | ☐ | FSIIHome | Home | Aug 28, 2005 |
      | 2 | ☐ | Home (fr) | Page Principale | Aug 16, 2006 |
      | 3 | ☑ | Home (de) | Home (de) | Aug 16, 2006 |
      | 4 | ☐ | Home (es) | Home (es) | Aug 16, 2006 |

   c. Click **Save**.

      The child page moves to the list of **Unplaced Pages** in the site tree.
4. Place the page asset under its new parent page:
   a. In the site tree, right-click on the placed page where you want to insert the unplaced child page, and choose **Place Page** from the pop-up menu.

      The place page form appears in the work area on the right.
   b. In the list of unplaced pages, type a number in the **Rank** field to designate the new page's position in the list of child page assets. Position numbering starts at 1, the top of the list. (In this example, it makes sense to type "1".)
   c. Click **Save**.

      The previously unplaced page asset moves to the site tree, to its assigned rank.

## Placing Page Assets and Workflow

CS-Direct has a workflow feature that controls the flow of assets as they pass from one team member to another; for example, from author to editor to approver to publisher. The workflow administrator can create processes that control who can place page assets in the site tree and during which workflow step they can do so. Note the following:

- The **Place Page** workflow privilege controls all place page functions: **Place Pages**, **Remove**, and **Rank**.
- You must have the proper privileges for both the parent page on which you invoke **Place Pages**, and for any child page that you want to **Rank** or **Remove**.

For information about the workflow process, see the *Content Server Administrator's Guide*.

## Editing Page Assets

In general, there are two ways to edit an existing page asset:

- Change the assets, but not the asset types, that are included on the page. For example, move new assets to the **Contains** list from your active list; select a different collection, query, or article from a named association field; or rebuild a collection already associated with the page asset to include different assets.
- Create a new association or change the actual structure of the page asset in some way.

Although you may frequently change the content in the collections or queries on a page at regular intervals, you are less likely to change the associations, asset types, or structure of a page after the site goes live. This may also require you to edit the code in the template element that formats the page.

## Deleting Page Assets

During your site design phase, it is likely that you will create and delete many page assets. However, before deleting a page asset from a site that you have published, be sure that you understand the consequences. For example:

- Have you removed references to the page from other page assets?
- Are any of your other page templates coded to extract and use information about this page asset in any way?

Before you delete a page asset, be sure to remove any references to it from any other elements or pages. It is a good idea to unplace a page asset before you delete it.

Chapter 26

# Best Practices for Creating Site Preview Assets and Templates

To successfully implement Site Preview functionality, content creators must understand how to create sets of time-sensitive assets. Developers must also include the tag that filters assets by date in templates that will display those assets.

This chapter contains the following sections:


- Considerations when Implementing Site Preview
- Creating Sets of Assets
- Writing Templates for Site Preview
- Caching Considerations

# Considerations when Implementing Site Preview

Using the Site Preview feature, content contributors can preview assets in the InSite interface as they will appear at a future time. This functionality utilizes asset "start date" and "end date" attributes. These two attributes determine the date range that assets are available on the web site. Content creators can specify start dates and end dates in "Edit" screens in both the Dash and Advanced interfaces. Instructions for assigning start/end dates are given in the Dash and Advanced interface user guides.

The following requirements must be met in order to properly implement the Site Preview feature:

- Content contributors must create an appropriate set of assets, and
- The administrator must update the templates that render the assets to include the tag `asset:filterassetsbydate`.

Each of these requirements is discussed in detail below.

# Creating Sets of Assets

When different versions of the same item will be displayed on different dates, content creators need to create multiple assets.

For example, suppose you are creating a page in a web site. You want to run a special version of the page for a one-day New Year's Day sale event. You would need three separate assets to accomplish this task. Here is how you would create the one-day sale:

- Create the regular page asset, setting its end date to the date before the sale day, December 31st 23:59:59.
- Create the sales event asset, setting the start date to the beginning of the sale day, January 1st 00:00:00, and the end date to the end of the sale day, January 1st 23:59:59.
- Duplicate the regular page asset, setting the start date to the day after the sale, January 2nd 00:00:00.

After the three assets are created, they can be passed as input to the `asset:filterassetsbydate` tag (discussed below), which will return the asset to render based on the given date.

For ease of use in searching for the related group of assets for editing, publishing, and filtering in templates, we recommend that developers designate an attribute (such as `name`) which content contributors fill in using a naming convention when creating related sets of time-sensitive assets.

# Writing Templates for Site Preview

Appropriate usage of the `asset:filterassetsbydate` tag is critical to Site Preview functionality. This tag filters assets according to a given date. The proper use of this tag is discussed below. More information about the `asset:filterassetsbydate` tag can be found in the tag reference.

## The asset:filterassetsbydate tag

The filter tag has the following format:

```
<asset:filterassetsbydate
   inputList="inputListName"
   outputList="outputListName"
   [date="date value in either yyyy-MM-dd HH:mm:ss OR yyyy-MM-dd
   format"]>
```

Notice that the filter tag has two input attributes (`inputList` and `date`) and one output attribute (`outputList`):

- The `inputList` attribute specifies the list of assets to be filtered based on the given date.
- The `date` attribute is optional. The `date` attribute is expected to be in either `yyyy-MM-dd HH:mm:ss` or `yyyy-MM-dd` format. The `date` attribute should be coded to accept the date value passed from the date picker in the InSite editing screen (use the `__insiteDate` variable for this).
- The tag produces an output list (`outputList`) which contains assets whose start/end dates enclose the given date.

The tag performs the following steps:

1. Checks for the presence of the InSite editor to determine if the template is operating on a content management system.
2. If the InSite editor property is turned off, the system date passes into the date field.

   If the InSite editor property is turned on, the tag routine:

   - Disables caching of the current page being rendered (see "Caching Considerations," on page 577 for more on caching),
   - Accepts the date parameter (passed from the InSite screen date picker), and
   - Checks for the appropriate format of the date passed into the tag.
3. Filters the input list of assets based on the given date to produce the output list.

## The Input List

The `asset:filterassetsbydate` tag requires the input list to contain two columns named `assetid` and `assettype`. This input list can be constructed in a variety of ways.

The simplest way to build the input list, and the way we recommend, is for content contributors to follow a naming convention when filling in the attribute of your choice (usually the `name` attribute). The list-building code could then be written using the `asset:search` tag or other similar tags to search that attribute for the agreed upon string and construct the list from the search results.

Another method for locating assets to place into the input list is to use the Recommendation asset type to hold a list of assets of interest.

Whichever method you use to determine the assets that need to be filtered, use the `listobject` tag to construct the input list for the filter tag, as shown in the sample code below.

This sample code creates a list object `inputListName` and adds a row containing two columns: `assetid` and `assettype`. The `listobject:tolist` then creates the input list called `assetInputList`. This list is now ready to be passed as input to the filter tag.

```
<listobject:create name="inputListName" columns="assetid,assettype" />
<listobject:addrow name="inputListName">
<listobject:argument name="assetid" value='<%=ics.GetVar("assetIdVar")%>' />
<listobject:argument name="assettype" value='<%=ics.GetVar("assetTypeVar")%>' />
</listobject:addrow>
<listobject:tolist name="inputListName" listvarname="assetInputList" />
```

Note that for the sake of simplicity, the code snippet explains the creation of an input list containing only one row. In practice users typically have multiple rows (usually read off from the results of `asset:search` tag or some other list) added to the list with each row representing an asset that needs to be filtered by a given date.

After creating the input list of assets to be filtered, use the `asset:filterassetsbydate` tag as follows:

```
<asset:filterassetsbydate inputList="assetInputList"
   outputList="assetOutputList"
   date='<%=ics.GetVar("dateValueVariable")%>' />
```

To pass input from the InSite interface date picker to the date attribute, replace the generic `dateValueVariable` with `__insiteDate`.

The tag produces an output list `assetOutputList`. Read through the list for assets that clear the filter by date test as follows:

```
<ics:if condition='<%=ics.GetList("assetOutputList")!=null &&
ics.GetList("assetOutputList").hasData()'%>
<ics:then>
   <ics:listloop listname="assetOutputList">
   <ics:listget listname="assetOutputList" fieldname="assetid"
output="id" />
   <ics:listget listname="assetOutputList" fieldname="assettype"
output="type"/>

<!--
Perform your usual asset load, asset get and other rendering
functions using Content Server tags here
-->
</ics:listloop>
</ics:then>
</ics:if>
```

# Caching Considerations

In a delivery system, in order for web pages to be displayed appropriately for the current date, asset entries must be removed from the cache at the proper times. Therefore, Content Server includes start and end dates in the factors it uses to calculate the expiry time for cached pages.

**Caution**

Content Server does not support the use of start and end dates with Export to Disk publishing. When assets are exported to disk, start and end date attributes are also exported to disk. However, the Export to Disk publishing method has no mechanism similar to the cache cleaning process (which, in other publishing methods, automatically removes expired assets from disk).

On a content management system, when a page is previewed in the InSite interface, the `asset:filterassetsbydate` tag disables page caching for that page. This ensures that the page being served always displays assets filtered by the date being passed from the InSite date picker, rather than serving pages from the page cache, which may have been generated using different date input.

Chapter 27

# Coding Elements for Templates and CSElements

Elements provide the code that identifies, extracts, and displays your content. In a Content Server system, your content is stored as assets. Therefore, much of the XML or JSP code in your elements is dedicated to identifying the appropriate asset for the appropriate context and then extracting and displaying that asset's data.

This chapter describes how to code the elements that you make for your template and CSElement assets. For information about creating the assets themselves, see Chapter 23, "Creating Template, CSElement, and SiteEntry Assets."

This chapter contains the following sections:


- About Dependencies
- About Coding to Log Dependencies
- Calling CSElement and SiteEntry Assets
- Coding Elements to Display Basic Assets
- About Coding Elements that Display Flex Assets
- Coding Templates That Display Flex Assets
- Creating URLs for Hyperlinks
- Handling Error Conditions

# About Dependencies

Your Content Server system tracks and relies on two kinds of dependencies to function correctly:

- **Approval dependencies**. These are conditions that determine whether an approved asset can be published.

  The approval system calculates the approval dependencies for an asset when it is approved. If there are dependent assets that also need to be approved, the parent asset will not be published.

- **Compositional dependencies**, that is, page composition dependencies. These are dependencies between assets and the pages and pagelets that they are rendered on that determine whether a page needs to be regenerated.

  The ContentServer servlet logs compositional dependencies when it renders pages. CacheManager consults the dependency log to determine when to regenerate the cached pages. The Export to Disk publishing method consults the dependency logs to determine when an exported page file must be regenerated.

**Note**

One of your responsibilities while coding elements is to ensure that your code logs compositional dependencies accurately, and, if you are designing a static site, that it sets approval dependencies appropriately, as well.

## The Publishing System and Approval Dependencies

The publishers, editors, and content providers who work on your management system **approve** assets to be published to a target destination. The publishing system then publishes the approved assets automatically, as a background process, according to the schedule that your administration team set up for your Content Server system.

An asset can be published only if it meets all specified approval dependencies, that is, all associated assets must have been either approved or previously published. If not, the asset is **held** from being published until the dependencies are met: the dependent (related) assets must themselves be approved for publishing to the same destination.

This approval process frees your content and editorial team from the responsibility of manually checking asset dependencies and then publishing a large number of related assets. It also ensures that there can be no broken links on your online site after assets are published.

If an asset is subsequently changed, the asset is no longer considered to be approved, and it must be approved again before it can be re-published.

### Calculating Approval Dependencies

Approval dependencies are recorded at the time the asset is approved. They are written to the `ApprovedAssetDeps` table in the Content Server database.

The approval status of an asset is determined by its dependency relationships, which include the approval status of all asset items associated with a particular asset item, as well as the dependency relationships of those associated items.

What is the basis for the dependency calculation? That depends on the publishing method:

- For **Export to Disk**, the approval system renders the asset using either the template that is assigned to it or, if there is one specified, the default approval templates for assets of this type. The tags in the template code set approval dependencies that determine the appropriate dependents for the approved asset. The dependent assets must be in an appropriate approval state before the current asset can be published.
- For **Mirror to Server** or **Export Assets to XML**, the approval process examines the data relationships between asset types. Basic assets have associations. Flex assets have family relationships. Both of these relationships create approval dependencies for these publishing methods. For example, if you approve a flex asset, it will be held from a publishing session unless its parent assets are in an appropriate approval state.

## Exists *vs.* Exact *vs.* None

Approval dependencies can be **exists**, **exact**, and **none**. This section defines each kind of approval type.

You cannot change the approval dependency type for CSElements and SiteEntry assets, embedded links and pagelets, or the Engage visitor data assets. With the exception of flex attributes—whose dependency type you set when you create the attribute—you also cannot change the approval dependency type for the flex family asset types. For basic asset types, you set the type of approval dependency for their associated assets when you configure the association fields.

When your publishing method is Export to Disk, the tags that set compositional dependencies when pages are rendered also create approval dependencies when the approval system calculates whether an asset can be published. When your code sets approval dependencies on pagelets generated for other assets, you can set the approval type to exists, exact, or none.

**Note**

For information about the types of approval dependencies created by the the relationships between assets of the various types, see the publishing chapter in the *Content Server Administrator's Guide*.

### Exists

With an **exists** dependency, the dependent asset must merely exist on the target—the version of the asset does not matter. An **exists** dependency means that an approved parent asset can be published even if a child asset changes (which means that the child asset is no longer approved), as long as the child asset was previously approved and published to that same destination.

For example, in the following sequence, a collection asset has an **exists** relationship with its ranked children:

1. A collection and all of its ranked articles are approved and published to a target.
2. One of the ranked articles is edited again, but not approved.
3. The collection itself is edited again, approved, and published to the destination.

   The collection is not held back from publishing by the changed but unapproved article, because a prior version of the article already "exists."

However, in the following example, a collection with an **exists** dependency relationship to its articles cannot be published:

1. A collection and all of its ranked articles are approved but not published.
2. One of the ranked articles is edited again.

Because the edited article was never published to the destination, it does not yet exist for that destination, which means that the collection cannot be published. The collection asset is **held** and both the collection and the edited article must be approved before the collection can be published.

The exists approval type is generally useful for links.

### Exact

With an **exact** dependency, the dependent asset must be the exact version on the target. No other previously approved version will do. An **exact** dependency means that the parent asset cannot be published if the version of the parent and child assets on the destination do not match.

In the following example, a page asset has an **exact** dependency with its article assets:

1. A page asset and all of its article assets are approved and published to a destination.
2. One of the articles is edited again, but is not re-approved.
3. The page asset is edited and is re-approved.

   The page asset is held, and the resulting form in the Content Server interface displays a link that points to a list of the assets that must be approved before the page asset can be published. This list shows the article that was edited but not re-approved.
4. The edited article is approved.

   The page asset has already been approved and can now be published because the version stamps of the article and the page asset match.
5. Another article asset associated with the page asset is edited.
6. Both the page asset and the edited article asset must be re-approved because the version stamps of the two do not match:
   - The article must be re-approved because it was edited but not yet re-approved.
   - The page asset must be re-approved because it was previously approved with a dependency on a different version of the article.

The exact approval type is generally useful for embedded content.

### None

A **none** dependency means that the approved asset can be published no matter what approval state the dependent asset is in. You can set the approval dependency type to `none` by adding the `DEPTYPE` parameter to a tag that sets an approval dependency and setting that parameter to `none`.

Note that setting `DEPTYPE` to `none` effects the approval dependency only. When the Export to Disk process generates the page and invokes the tag, a compositional dependency is logged. But when the approval system invokes the tag during its calculation, no approval dependency is logged.

## Approval Templates for Export to Disk

When assets are approved for a publishing destination that uses the Export to Disk publishing method, the approval system examines the template assigned to the asset to determine its dependencies.

However, when Export to Disk actually publishes the asset, it does not necessarily use the template that is assigned to the asset. Why? Because the code in another element could determine that a different template is used for that asset in certain cases.

Consider the Burlington Financial sample site. An article asset from this sample site can be rendered by several different templates, depending on the context.

So when you approve an article asset for the Burlington Financial sample site, which template should the approval process use to determine the dependencies for the article? The one that contains the most representative set of dependencies for all of the templates. For an example, see the Burlington Financial template article named Full. You may decide to create a special template that contains all the possible dependencies for assets of each type.

What if the template that contains the most representative set of dependencies is not the template that you want to assign to the asset? Set it as the **Default Approval Template** for assets of that type.

You can set Default Approval Templates for each asset type and for each publishing destination. This feature is located in the tree on the **Admin** tab:

**Default templates for: Destination 1 (static)**

| Asset Type | Subtype | Select a Template | Wrapper page |
|---|---|---|---|
| **Article** | Columnist | no default, use asset's template field | None |
| | Standard | no default, use asset's template field | None |
| **Article Flex** | DrillHierarchyCT | no default, use asset's template field | None |
| | Story | no default, use asset's template field | None |
| **CSElement** | (no subtype) | no default, use asset's template field | None |
| **Collection** | Article | no default, use asset's template field | None |
| | AArticles | no default, use asset's template field | None |
| | Collection | no default, use asset's template field | None |
| | Content_C | no default, use asset's template field | None |
| | ContentGroups | no default, use asset's template field | None |
| | Content_P | no default, use asset's template field | None |
| | Dimension | no default, use asset's template field | None |
| | Document_C | no default, use asset's template field | None |
| | FSIIVisitor | no default, use asset's template field | None |
| | FSIIVisitorParent | no default, use asset's template field | None |
| | StyleSheet | no default, use asset's template field | None |
| **Recommendation** | (no subtype) | no default, use asset's template field | None |
| **SiteEntry** | (no subtype) | no default, use asset's template field | None |

Edit Configuration

**Note**

If you specify a default approval template for an asset type on a destination that uses the Mirror to Server publishing method, that template is used when you preview the asset on the Asset Status screen, but not when the asset is approved or published.

### Subtypes, Flex Definitions, and Approval Templates

If you are using flex assets for a static site, you can assign more than one default approval template to the flex asset type in the family. You can designate a different default approval template for each flex definition.

For basic assets, the Subtype feature provides a way to further categorize assets of a single asset type. You can use this feature to assign more than one default approval template for assets of a specific type, based on some other organizing construct.

For example, perhaps the approval template for sports articles should be different than the approval template for world news articles. You can create a sports subtype and a world news subtype for the article asset type and then assign different approval templates for each subtype of the asset type.

You create subtypes for basic assets either in the asset descriptor file when you create the asset type or by using the **Asset Types** option on the **Admin** tab if you decide you need subtypes after the asset types were created. You assign a subtype to an asset by using the New and Edit asset forms. As mentioned, flex assets already have subtypes: their flex definitions.

For more information about configuring subtypes for basic assets, and about subtypes in general, see Chapter 15, “Designing Basic Asset Types.”

## Page Generation and Compositional Dependencies

Compositional dependencies are recorded in different ways:

- When the Export to Disk publishing method renders a page, it logs compositional dependencies to the appropriate publishing tables. Then, when it’s time to publish again, Export to Disk can determine which pages need to be regenerated based on which assets are being published—it generates all the pages that have logged the assets as compositional dependents.
- When Content Server renders and caches a page, it logs the dependencies in the `SystemItemCache` table at the time a page is rendered and cached. Each row in this table holds the ID of an asset and the cache key or ID of the generated page that the asset was rendered on.

### CacheManager and the Page Caches

The CacheManager maintains the Content Server page caches. As assets are changed, it consults the `SystemItemCache` table to determine which cached pages those assets were rendered on. Then it works through the `SystemPageCache` table, flushing and regenerating the appropriate pages.

After it makes changes to the Content Server page cache, CacheManager communicates that information to all the Satellite Servers participating in your Content Server system, the co-resident Satellite Server and any remote Satellite Servers that are installed in your system. The Satellite Server applications then update the Satellite page caches.

**Note**

If you have the appropriate permissions, you can examine the data in the `SystemItemCache` and `SystemPageCache` tables, but, as with any other system table in the Content Server database, do not alter the information stored in these tables in any way.

### CacheManager and Dynamic Publish Sessions

The CacheManager interacts with the publishing system during Mirror to Server publishing session. When a Mirror to Server publishing session ends, the publishing system provides a list of all the IDs of all the assets that were included in the publish operation to the CacheManager servlet on the destination system.

The CacheManager compares that list to the compositional dependencies logged for the pages in the cache to determine which pages and pagelets need to be flushed from the page cache and regenerated. It updates the Content Server page cache accordingly, and then sends the list of pages to the co-resident and remote Satellite servlets so they can flush those same pages and get new versions from the Content Server page cache.

### CacheManager and the Preview Function

When you preview an asset (on the development or management system), the Content Server interface executes the page name of the template for the asset. ContentServer renders the page, caches the page, and logs the compositional dependencies between the rendered page and the asset.

CacheManager updates the cached versions of previewed pages when assets are saved. That is, when someone clicks **Save**, CacheManager compares the object ID of that asset to the compositional dependencies logged for the pages in the cache. It then clears and refreshes the appropriate pages in the page cache and communicates the information about the changed pages to the Satellite servlets.

## About Coding to Log Dependencies

While you are coding elements, one of your responsibilities is to include code that logs dependencies accurately.

There are several tags that log compositional dependencies. When the tag is executed, Content Server logs a dependency between the rendered page and the asset by writing this information in the `SystemItemCache` table.

Note that for a static site using the Export to Disk publishing method, the tags that log compositional dependencies can also log approval dependencies. When an asset is approved, the approval system renders that asset to determine whether it can be published. It logs the results of these tags to the `ApprovedAssetDep` table unless the tag sets the approval dependency type to none. (See "Exists vs. Exact vs. None" on page 581 for more information about the "none" dependency type.)

This section presents the tags that log dependencies alphabetically. For more information about these and any other tag, see the *Content Server Tag Reference*.

## ASSET.LOAD and asset:load

When Content Server executes an `ASSET.LOAD` tag (or `asset:load`), it automatically logs a compositional dependency for the asset that is loaded. For example:

```
<ASSET.LOAD TYPE="Page" NAME="target" FIELD="name" VALUE="Home"/>
```

That line of code marks a compositional dependency between the page asset named "Home" and the rendered page that is displaying this asset.

### Setting the Approval Dependency Type

When an asset is approved for an Export to Disk destination and the approval system renders this tag, the tag also logs an approval dependency between the assets that are in play.

By default, the approval dependency for `ASSET.LOAD` is set to `exact`. You can set the dependency to `exists` or to `none` by using the `DEPTYPE` parameter. For example:

```
<ASSET.LOAD TYPE="Page" NAME="target" FIELD="name" VALUE="Home"
DEPTYPE="exists"/>
```

## The ASSETSET (assetset) Tag Family

You use the `ASSETSET` tag family to create a set of one or more flex assets. The following tags create assetsets and define compositional dependencies for the assets in the set:

```
ASSETSET.SETASSET and assetset:setasset
ASSETSET.SETEMPTY and assetset:setempty
ASSETSET.SETLISTEDASSETS and assetset:setlistedassets
ASSETSET.SETSEARCHEDASSETS and assetset:setsearchedassets
```

When an asset from the assetset is rendered, the compositional dependency is logged.

The first three tags define the following compositional dependencies:

- A dependency between each flex asset in the assetset and the rendered page.
- A dependency between the flex asset's parents and the rendered page. Because flex assets inherit values from their flex parent assets, a change to a parent can mean a change to the flex asset and that means the pages that hold the asset may no longer be accurate.

The fourth tag, `assetset:setsearchedassets`, creates an assetset from the results of a search state. Search states are queries, which means there is no way to predict the identities of the assets in the set. Therefore, the `ASSETSET.SETSEARCHEDASSETS` tag defines the compositional dependency as "unknown." When a compositional dependency is unknown, it means the page must be regenerated during each Export to Disk publishing session and updated in the page caches after each Mirror to Server publishing session, whether it needs it or not.

If you have a search state that describes a fixed set of assets whose identities will not change, you instruct Content Server to set compositional dependencies for the assets in the assetset by setting the optional `fixedlist` property to "true."

For example:

```
<assetset:setsearchedassets name="as" assettypes="Products"
constrain="ss" fixedlist="true" />
```

This example defines that there is a compositional dependency between each product asset in the assetset named “as” and the rendered page.

For more information about asset sets and search states, see “Assetsets” on page 599 and “Searchstates” on page 600.

### Setting the Approval Dependency Type

If you are using flex assets for a static site, be aware that when the approval system invokes an assetset tag, the approval dependency type is set to `none` by default. To change this value to exists or exact, you use the `deptype` parameter.

For example:

```
<assetset:setsearchedassets name="as" assettypes="Products"
constrain="ss" fixedlist="true" deptype="exists" />
```

Note that setting an approval type for the `assetset:setsearchedassets` tag is meaningful only if the `fixedlist` parameter is set to `true`.

## RENDER.GETPAGEURL and render:getpageurl

The `RENDER.GETPAGEURL` tag creates a URL for assets that are not blobs. This tag logs an exists **approval** dependency—but **not** a compositional dependency—between the asset being approved (rendered) and the asset referred to by the tag. This means that it creates a dependency only when your publishing method is Export to Disk.

In this example, the template assigned to article ABC has the following code in it:

```
<RENDER.GETPAGEURL PAGENAME="BurlingtonFinancial/Page/Home"
            cid="Variables.pageid"
            c="Page"
            OUTSTR="referURL"/>
```

That code fragment both creates a URL (that is returned in the variable created by the `OUTSTR` parameter) and logs an **exists** approval dependency between the asset identified in the `cid` variable and article ABC.

Then, when article ABC is approved, the page identified by the `cid` variable must either be approved or must already have been published or article ABC is held from being published.

## RENDER.LOGDEP (render:logdep)

There are several situations in which your code can obtain an asset’s data without actually loading the asset. When this is the case, be sure to log the compositional dependency yourself with the `render:logdep` tag.

### Example 1

When you call a CSElement from a Template asset or other CSElement asset, you do not load the asset to determine the identity of the element file to execute. Instead, you use the `RENDER.CALLELEMENT` or `render:callelement` tag and invoke the element directly by name. For example:

```
<render:callelement name="BurlingtonFinancial/Common/
HeaderText"/>
```

Because you didn't use the `asset:load` tag to access the CSElement, the compositional dependency between the CSElement asset and the page it is being rendered on is not automatically logged for you. Instead, you must set it yourself.

At the beginning of the element for each CSElement asset, you include the following line of code:

```
<render:logdep cid="Variables.eid" c="CSElement"/>
```

At the beginning of the element for a Template asset, the `render.logdep` statement would be as follows:

```
<render:logdep cid="Variables.tid" c="template"/>
```

Note that if you use the "CSElement" form or the template form in the Content Server interface to start coding the element, Content Server automatically includes an appropriate `render:logdep` statement in the stub code that it seeds into the element for you.

### Example 2

For basic assets, when you use an `ASSET.LOAD` tag on a parent asset (basic asset) and then use an `ASSET.CHILDREN` tag, you have access to the children assets' data without having to load it. In this case, you should include a `RENDER.LOGDEP` statement to log the compositional dependency.

For example:

```
<ASSET.CHILDREN NAME="PlainListCollection" LIST="theArticles"
      OBJECTTYPE="Article" ORDER="nrank" CODE="-"/>
<LOOP LIST="theArticles">
      <RENDER.LOGDEP cid="theArticles.id" c="Article"/>
      .
      .
      .
```

## Setting the Approval Dependency Type

When an asset is approved for an Export to Disk destination and the approval system invokes this tag, the tag also creates an exact approval dependency between the asset and the rendered page.

You can change the approval dependency type to exists or none by setting the `DEPTYPE` argument. For example:

```
<RENDER.LOGDEP cid="theArticles.id" c="Article"
DEPTYPE="exists"/>
```

# RENDER.FILTER and render:filter

You use the `RENDER.FILTER` tag for lists of assets created by queries. This tag filters out any unapproved assets from a list or a query. It also sets a compositional dependency of "unknown." (The "unknown" compositional dependency is explained in the next section, "RENDER.UNKNOWNDEPS and render:unknowndeps.")

You use this tag when you do not want an approved asset that has an approval dependency on the results of a query (a collection or query asset, for example) to be held from being published when there are unapproved assets in the list that is returned by the query. For example, say that the element is coded to provide appropriate formatting for any number of article assets that are passed to it so it doesn't matter if only two of the five articles

included in a collection cannot be published. Because this tag tells Export to Disk to filter out the unapproved assets, a page using the query can be published while the unapproved assets remain unpublished.

You might use this tag in the following places:

- Templates for query assets
- Templates for collection assets
- `SELECTTO` statements and `EXECSQL` queries

For example:

```
<RENDER.FILTER LIST="ArticlesFromWireQuery"
LISTVARNAME="ArticlesFromWireQuery" LISTIDCOL="id"/>
```

## RENDER.UNKNOWNDEPS and render:unknowndeps

The `RENDER.UNKNOWNDEPS` tag signals that there are dependent assets but that there is no way to predict the identities of those assets because they came from a query or change frequently. This tag logs a compositional dependency of "unknown" for the rendered page. This tag does not set an approval dependency for the Export to Disk publishing method.

When a compositional dependency is set to "unknown," it means the page must be regenerated during each Export to Disk publishing session and updated in the page caches after each Mirror to Server publishing session, whether it needs it or not.

**Note**

You must use this tag carefully because the more pages that must be regenerated, the longer it takes to publish your site.

You use this tag to cover those coding situations in which you truly cannot determine what the dependent assets might be. For example, queries are dynamic and can retrieve a different resultset every time they are run. When you use queries of any kind—query assets, `SELECTTO` statements, `EXECSQL`, and so on—you should use the `RENDER.UNKNOWNDEPS` tag.

# Calling CSElement and SiteEntry Assets

When your design requires that your code call a CSElement or SiteEntry asset, there is no need to load the asset itself. From a coding point of view, you are not interested in the CSElement or SiteEntry as an asset—you are solely interested in the element or page entry that the asset represents. Therefore, your code can directly invoke the element or page entry with the appropriate tag.

If a CSElement does not have a corresponding SiteEntry asset (which means its output is cached according to the cache criteria set for the calling page), or, if you don't need a separate pagelet at this invocation, you invoke it by name with the `RENDER.CALLELEMENT` (`render:callelement`) tag. For example:

```
<render:callelement name="BurlingtonFinancial/Common/
SetHTMLHeader"/>
```

When CSElement does have a corresponding SiteEntry asset, you invoke the element by calling the page name of its SiteEntry asset with the `RENDER.SATELLITEPAGE` (`render:satellitepage`) tag. For example:

```
<render:satellitepage pagename="BurlingtonFinancial/Pagelet/
Common/SiteBanner"/>
```

**Note**

When you use CS-Explorer to examine `SiteCatalog` and `ElementCatalog` entries, they are presented as folders and subfolders that visually organize the pages and pagelets. However, these entries are simply rows in a database table—there is no actual hierarchy. Therefore your code must always call a page entry or an element entry by its entire name. You cannot use a relative path.

Additionally, the chain of called elements should not be more than 20 levels deep. Otherwise, the system will perform poorly when displaying the assets.

Also, if you edit using CS-Explorer, save the asset in the asset's editorial form (in the Content Server interface) to ensure that the cache is updated to reflect your edits. (CS-Explorer does not automatically update the cache.)

# Coding Elements to Display Basic Assets

To code an element for a template, you need to understand how the asset type it formats is designed. Having a preliminary understanding of data design and site design will prevent you from having to recode templates in order to re-display assets when they are updated. (That is why this developer's guide covers both data design and site design in the same book.)

For example, the Home page asset for the Burlington Financial site has four collections and one query assigned to it through named associations:

- TopStory collection
- SideBarTop collection
- SideBarMiddle collection
- SideBarBottom collection
- Wirefeed Query

The Home page asset has a template, also called `Home`. The `Home` template is coded to identify the collections and the query related to the Home page through these named associations and to display the assets in the collection and the assets returned by the query.

Because the `Home` template is coded to handle any collection or query that is associated with the Home page through these named associations (rather than hard-coded to extract specific articles), the assets that are displayed on the page can be updated as often as necessary but the code does not need to be changed.

Content providers can change the articles in the collections, and the wire feed service can make daily updates to the articles that the Wirefeed Query obtains. And no matter which

articles are selected in the collections or returned by the query, they are always formatted in the same way.

This section provides:

- Information that you should keep in mind while you code templates for basic asset types.
- Code fragments and examples for various situations, including managing the dependencies between assets so that approval can be calculated correctly for static sites and so that the page cache can be cleared when appropriate for dynamic sites.

Before you begin, be sure to read the chapters in the Programming Basics section of this book, especially Chapter 4, "Programming with Content Server."

For information about the tags used in the code examples in this chapter, see the *Content Server Tag Reference*.

For more code samples that display basic assets, see Chapter 28, "Template Element Examples for Basic Assets."

## Assets That Represent Simple Content

Template elements for content assets generally extract one specific article, advertising copy, special offer, image, and so on from the database, then obtain information from the relevant fields such as headline, body, and byline (for example), and then display that information online.

Consider the following simple template element designed for a Burlington Financial article asset:

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Article/VeryBasic
-
- INPUT
- Variables.c - asset type (Article)
- Variables.cid - id of the asset to display
- Variables.tid - template used to display the page(let)
- OUTPUT
-
-->
<!-- log the template as a dependent of the pagelet being
rendered, so changes to the template will force regeneration of
the page(let) -->

<IF COND="IsVariable.tid=true">
   <THEN>
   <RENDER.LOGDEP cid="Variables.tid" c="Template"/>
   </THEN>
</IF>
<!-- asset load will mark the asset as an 'exact' dependent of the
pagelet being rendered -->

<ASSET.LOAD NAME="anAsset" TYPE="Variables.c"
OBJECTID="Variables.cid"/>
```

```
<!-- get all the primary table fields of the asset -->

<ASSET.SCATTER NAME="anAsset" PREFIX="asset"/>

<!-- display the description -->
<ics.getvar name="asset:description"/>

<!-- display the contents of the urlbody file -->

<ics.getvar name="asset:urlbody" encoding="default"
   output="bodyvar"/>
<RENDER.STREAM VARIABLE="bodyvar" /><br/>

</FTCS>
```

This code in this template does the following things:

- Logs a compositional dependency between the Template asset and the page being rendered with the element with the `RENDER.LOGDEP` tag.
- If the approval system is evaluating this code for an Export to Disk target, logs an approval dependency.
- Loads the article asset with an `ASSET.LOAD` tag, which logs a compositional dependency between the article asset and the page being rendered.
- Extracts all the values from all the fields of the article with an `ASSET.SCATTER` tag.
- Displays the contents of the description column with a `CSVAR` tag. The description column corresponds to the **Headline** field in the **New** or **Edit** article forms in the Content Server interface.
- Displays the contents of the urlbody column with the `ics.getvar` and `RENDER.STREAM` tags. The urlbody column corresponds to the **Headline** field in the **New** or **Edit** article forms in the Content Server interface.

Notice the difference in the code that displays the value from `description` column and the code that displays the value from the `urlbody` column. The `urlbody` column can contain embedded links and whenever a field can contain embedded links, you ensure that the links are rendered correctly by using the `RENDER.STREAM` tag rather than the `CSVAR` tag.

For a more complex example of an article template, examine the Burlington Financial template named Full. You can examine it in two ways:

- Search for and then inspect it in the Content Server interface.
- Use Content Server Explorer to open the template element called:

  `ElementCatalog/BurlingtonFinancial/Article/Full.`

This template element provides the format for an article when it is displayed, in full, on a page in a browser.

## Associations

You identify the assets that are associated with other assets through association fields with the `ASSET.CHILDREN` tag. To specify which associated asset, you use the `CODE` parameter to specify the association field.

For example, say that the following code fragment is inserted right before the `</FTCS>` tag in the preceding example:

```
<!-- display the Main Image -->
<ASSET.CHILDREN NAME="anAsset" LIST="associatedImage"
   CODE="MainImage"/>
<IF COND="IsList.associatedImage=true">
   <THEN>
   <RENDER.SATELLITEPAGE PAGENAME="BurlingtonFinancial/ImageFile/
   TeaserSummary" ARGS_cid="associatedImage.oid"/>
   </THEN>
</IF>
```

The code in this fragment does the following things:

- Extracts the imagefile asset that is specified in the Main Image field for this article asset (named "anAsset") with the `ASSET.CHILDREN` tag and the `CODE` parameter set to "MainImage."
- Passes the identity of that imagefile to the page entry for the TeaserSummary template with the `RENDER.SATELLITEPAGE` tag. The page entry is identified with the `PAGENAME` parameter and the imagefile is identified with the `ARGS_cid` parameter. The TeaserSummary template than renders the imagefile into a pagelet and passes the pagelet back to this page, where it is displayed with the article.

## ImageFile Assets or Other Blob Assets

The imagefile asset type stores uploaded image files. In other words, the imagefile asset type is a **b**inary **l**arge **ob**ject (blob), served from the Content Server database. You use the BlobServer servlet to serve and display imagefiles and other blobs.

A template element for an imagefile or other blob can use the `RENDER.SATELLITEBLOB` tag to create and return an HTML tag that tells the browser how to access the blob and how to format and display it. If you need a BlobServer URL only, without it being embedded in an HTML tag, you can use the `RENDER.GETBLOBURL` tag.

For more information about coding links to blobs, see "Creating URLs for Hyperlinks" on page 613.

## Basic Assets That Can Have Embedded Links

The **Body** field of the Article asset and other assets that have fields with a data type of `TEXTAREA` allow editors to create embedded hyperlinks within the text field. To ensure that these links are rendered properly, use the `RENDER.STREAM` tag to retrieve the contents of the field, as shown in the following example:

```
<ASSET.LOAD TYPE="Variables.c" OBJECTID="Variables.cid"
   NAME="TestArticle"/>
<ASSET.SCATTER NAME="TestArticle" PREFIX="articleAsset"/>

<!-- display the contents of the urlbody file -->

<ics.getvar name="articleAsset:urlbody" encoding="default"
output="bodyvar"/>
<RENDER.STREAM VARIABLE="bodyvar" /><br/>
```

If InSite Editor is enabled on your management system, note that the `INSITE.EDIT` tag also manages embedded links appropriately when it retrieves the contents of a field that has embedded links in it. For more information about the InSite Editor, see Chapter 35, "Coding for the InSite Editor."

## Collections

Templates for collection assets typically extract the assets in the collection from the database with an `ASSET.CHILDREN` tag. For example:

```
<ASSET.LOAD TYPE="Variables.c" OBJECTID="Variables.cid"
   NAME="PlainListCollection"/>
<ASSET.SCATTER NAME="PlainListCollection" PREFIX="asset"/>
<ASSET.CHILDREN NAME="PlainListCollection" LIST="theArticles"
   OBJECTTYPE="Article"/>
```

After the children are identified, the template code can then display parts of these assets in a list on a rendered page.

Sometimes the template for a collection is coded to handle the first item in the collection differently than the rest. You can single out the highest ranking asset in a collection by coding the element to order the items in the list according to their rank, as shown here:

```
<ASSET.CHILDREN NAME="HomePageStories" LIST="theArticles"
   OBJECTTYPE="Article" ORDER="nrank"/>
```

For a longer example, examine the Burlington Financial template named MainStory List. You can examine it in two ways:

- Search for and then inspect it in the Content Server interface.
- Use Content Server Explorer to open the template element called:

  `ElementCatalog/BurlingtonFinancial/Collection/MainStoryList`

This template element calls two page entries for two other templates. The root element for the first of the two page entries displays the highest ranked article from the collection. The root element for the second displays the rest of the articles.

### Collection Templates and Approval Dependencies

When your publishing method is Export to Disk, you can use the `RENDER.FILTER` tag in your collection templates. This tag filters out any unapproved assets from the collection both when the approval dependencies are calculated and when the publish process renders the site.

The following code fragment, taken from the Burlington Financial StoryList template, illustrates this tag:

```
<ASSET.LOAD TYPE="Variables.c" OBJECTID="Variables.cid"
   NAME="StoryListCollection"/>
<ASSET.SCATTER NAME="StoryListCollection" PREFIX="asset"/>
<ASSET.CHILDREN NAME="StoryListCollection" LIST="theArticles"
   ORDER="nrank" CODE="-"/>

<!-- Get only the articles that are approved for export -->

   <RENDER.FILTER LIST="theArticles"
```

```
         LISTVARNAME="ApprovedArticles"
         LISTIDCOL="oid"/>

<!-- Display only the articles that are approved-->

      <IF COND="IsList.ApprovedArticles=true">
         <THEN>
            <LOOP LIST="ApprovedArticles">
            <RENDER.SATELLITEPAGE
                  PAGENAME="BurlingtonFinancial/Article/Summary"
                  ARGS_cid="ApprovedArticles.oid"
                  ARGS_p="Variables.p"/>
            </LOOP>
         </THEN>
      </IF>
```

## Collection Templates and Compositional Dependencies

In the preceding code example that illustrates the `RENDER.FILTER tag`, the ID of each of the child assets in the collection is passed to the Summary template.

The first line of code in the Summary template is an `ASSET.LOAD` statement, which means that the dependency between article asset that it loads and the page that is rendered with the Summary template is logged.

What if the code in the template for the collection also formats the child articles? In that case, you must carefully consider the code and determine whether you need to log the dependency with the `RENDER.LOGDEP` tag.

For example, when you use the `OBJECTYPE` parameter in an `ASSET.CHILDREN` tag, the resulting list is a join of the `AssetRelationTree` table and the asset table for the type specified and includes information from both tables. For example

```
<ASSET.CHILDREN NAME="StoryListCollection" LIST="theArticles"
   OBJECTTYPE="Article" ORDER="nrank" CODE="-"/>
```

You can then access the children asset's information without using subsequent `ASSET.LOAD` tags. If you do, be sure to include the `RENDER.LOGDEP` tag for each child so that the compositional dependencies between those assets and the rendered page can be tracked correctly.

For another example, see "Example 2: Coding Links to the Article Assets in a Collection Asset" on page 625.

# Query Assets

Query assets can execute SQL code or they can run an element that contains query code. You use them in collections, on page assets, and so on:

- You build a collection by running a query in the "Build Collection" form and then selecting and ordering the assets you want from the resulting list. The collection is a static list of assets selected from the query's resultset.
- You select queries for a page asset either through unnamed relationships or through named associations. You select queries for assets like articles through named associations.

In these cases, the page or article assets do not themselves invoke the query: you code the query template element to invoke a standard CS-Direct element called `OpenMarket/Xcelerate/AssetType/Query/ExecuteQuery`. This element runs the query asset when the page asset or article asset is rendered.

Elements for query templates invoke the `ExecuteQuery` element and typically include code that loops through the items returned in the list object that the query created, extracts bits of information from those items, and then displays it.

The following example loads a query asset and passes it to the `ExecuteQuery` element:

```
<ASSET.LOAD TYPE="Query" NAME="Wirefeed"
OBJECTID="Variables.id"/>
<CALLELEMENT NAME="OpenMarket/Xcelerate/AssetType/Query/
                  ExecuteQuery">
         <ARGUMENT NAME="list" VALUE="ArticlesFromWireFeed"/>
         <ARGUMENT NAME="assetname" VALUE="WireFeed"/>
         <ARGUMENT NAME="ResultLimit" VALUE="-1"/>
</CALLELEMENT>
```

For a longer example, examine the Burlington Financial query template named `PlainList`. You can examine it in two ways:

- Search for and then inspect it in the Content Server interface.
- Use Content Server Explorer to open the template element called:

  `ElementCatalog/BurlingtonFinancial/Query/PlainList.`

This element invokes the `ExecuteQuery` element to run the PlainListQuery query asset, filters out any unapproved asset if the publishing method is Export to Disk, and then loops through the resulting list, obtaining a dynamic URL for each item in the list and creating a hyperlink for it.

For information about hyperlinks, see "Creating URLs for Hyperlinks" on page 613.

## Queries and Compositional Dependencies

The first line of code in the `ExecuteQuery` element is a `RENDER.UNKNOWNDEPS` tag, which alerts the Export to Disk publishing method and the CacheManager on a dynamic delivery system that the assets that will be retrieved by the query cannot be predicted and, therefore, no dependencies can be calculated and logged.

If you are using any other kind of query—for example, a `SELECTTO` statement, `CALLSQL`, or `EXECSQL`—you should include the `RENDER.UNKNOWNDEPS` tag.

Additionally, in the element that a query-generated list of assets is returned to, you must use the `RENDER.FILTER` tag if you are using the Export to Disk publishing method. For example:

```
<CALLELEMENT NAME="OpenMarket/Xcelerate/AssetType/Query/
ExecuteQuery">
   <ARGUMENT NAME="list" VALUE="ArticlesFromTheQuery"/>
   <ARGUMENT NAME="assetname" VALUE="PlainListQuery"/>
   <ARGUMENT NAME="ResultLimit" VALUE="5"/>
</CALLELEMENT>

<!-- On export - filter out un-approved assets -->
<RENDER.FILTER LIST="ArticlesFromTheQuery"
LISTVARNAME="ArticlesFromTheQuery" LISTIDCOL="id"/>
```

```
<if COND="ArticlesFromTheQuery.#numRows!=0">
<then>
         <LOOP LIST="ArticlesFromTheQuery">
            <RENDER.GETPAGEURL PAGENAME="BurlingtonFinancial/
   Article/
            Variables.ct"
                  cid="ArticlesFromTheQuery.id"
                  c="Article"
                  p="Variables.p"
                  OUTSTR="referURL"/>
            <A class="wirelink" HREF="Variables.referURL"
            REPLACEALL="Variables.referURL"><ics.listget
   listname="ArticlesFromTheQuery" fieldname="subheadline"/>
   </A><P/>
```

For another example, see "Example 4: Coding Templates for Query Assets" on page 631.

## Page Assets

Templates for page assets generally contain the following kinds of code:

- The framework for the page asset when it is a rendered page
- The logic for obtaining the content for the rendered page
- The logic for links to other rendered pages

The templates for content assets contain the formatting code for individual pieces of content. The page templates invoke the templates for the other assets, receive formatted assets from those template elements, and then place the formatted assets into the context of the page framework.

Following is the code for a simple template that formats a page asset:

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Page/CollectionsAndQuery
-
- INPUT
- Variables.c - asset type (Page)
- Variables.cid - id of the asset to display
- Variables.tid - template used to display the page(let)
- OUTPUT
-
-->

<!-- log the template as a dependent of the pagelet being
rendered, so changes to the template will force regeneration of
the page(let) -->

<IF COND="IsVariable.tid=true">
   <THEN>
   <RENDER.LOGDEP cid="Variables.tid" c="Template"/>
   </THEN>
```

```
</IF>

<!-- asset load will mark the asset as an 'exact' dependent of the
pagelet being rendered -->

<ASSET.LOAD NAME="anAsset" TYPE="Variables.c"
   OBJECTID="Variables.cid"/>

<!-- get all the primary table fields of the asset -->

<ASSET.SCATTER NAME="anAsset" PREFIX="asset"/>

<!-- get a list of id's of the child assets in the collection in
order of their rank -->

<!-- get the WireFeed query -->

<ASSET.CHILDREN NAME="HomeTextPage" LIST="WireFeedStories"
   CODE="WireFeed"/>
<IF COND="IsList.WireFeedStories=true">
   <THEN>
   <RENDER.GETPAGEURL PAGENAME="BurlingtonFinancial/Query/
   WireFeedFrontText"
         cid="WireFeedStories.oid"
         c="Query"
         p="Variables.asset:id"
         OUTSTR="referURL"/>
   <P>
   <A HREF="Variables.referURL"
         REPLACEALL="Variables.referURL">From the Wires...</A>

   </P>
   <RENDER.SATELLITEPAGE PAGENAME="BurlingtonFinancial/Query/
   WireSummaryText"
         ARGS_cid="WireFeedStories.oid"
         ARGS_ct="WireStoryText"
         ARGS_p="Variables.asset:id"/>
   </THEN>
</IF>
</FTCS>
```

The code in this example does the following things:

- Logs a compositional dependency between the Template asset and the page being rendered with a `RENDER.LOGDEP` tag.
- Loads the page asset with an `ASSET.LOAD` tag, which logs a compositional dependency between the article asset and the page being rendered.
- Extracts the WireFeed query with an `ASSET.CHILDREN` tag and the `CODE` parameter set to “WireFeed.”
- Obtains a URL for a page that will display the stories from the WireFeed query with the `RENDER.GETPAGEURL` tag. The `PAGENAME` parameter specifies the page entry of the template to use to create that page and also determines part of the URL. The

OUTSTR parameter creates a variable named referURL to hold the URL that RENDER.GETPAGEURL creates.

- Uses the URL from the referURL variable to build an <A HREF> link to the page.
- Passes the identity of the query asset to the page entry for the WireSummaryText template. The WireSummaryText template then creates a pagelet that displays the summary text from each article returned by the Wire Feed query and passes the pagelet back to this page, where it is displayed.

For a more complex example of a page asset template, examine the Burlington Financial template named SectionFront. You can examine it in two ways:

- Search for and then inspect it in the Content Server interface.
- Use Content Server Explorer to open the template element called:

  ElementCatalog/BurlingtonFinancial/Page/SectionFront

This element creates a Section Front page with a navigational bar on the top, a navigational area with links on the left, a list of stories, and so on.

# About Coding Elements that Display Flex Assets

When you code templates for basic assets, you use the CS-Direct ASSET tag family. For example, when you want to extract and display a basic asset, you use the ASSET.LOAD tag, a tag that extracts data from the primary storage table for that asset type.

Because the database schema for flex assets is different than that for basic assets, CS-Direct Advantage provides additional tag families for flex assets that you use in place of the ASSET tags:

- ASSETSET. You use this tag family to specify a set of one or more flex assets.
- SEARCHSTATE. You use this tag family to create search constraints that filter the assets in an assetset.

**Note**

The ASSET.LOAD tag will load a flex asset for you. However, using the ASSET.LOAD tag with flex assets is not supported: the code cannot be upgraded, and extracting the asset in this way is slower by orders of magnitude than using the ASSETSET tag family.

When you use the flex asset model to represent your content, your online site will use a mixture of flex and basic assets because the page asset type (which you are likely to use) is a basic asset type.

## Assetsets

An **assetset** is a group of one or more flex assets or flex parent assets. You use the ASSETSET tags to create the set of assets and to extract the attribute values that you want to display.

You can retrieve the following information from an assetset:

- The values for one attribute for each of the flex assets in the assetset.

- The values for multiple attributes for each of the flex assets in the assetset.
- A list of the flex assets in the assetset
- A count of the flex assets in the assetset
- A list of unique attribute values for an attribute for all flex assets in the assetset
- A count of unique attribute values for an attribute for all flex assets in the assetset

You can create assetsets that include flex assets of more than one type, but only if those flex assets use the same flex attribute asset type.

The most commonly used `ASSETSET` tags are:

```
ASSETSET.SETASSET
ASSETSET.SETSEARCHEDASSETS
ASSETSET.GETMULTIPLEVALUES
ASSETSET.GETATTRIBUTEVALUES
ASSETSET.GETASSETLIST
ASSETSET.SORTLISTENTRY ...
```

All of the `ASSETSET` tags are described in the *Content Server Tag Reference* and several of them are used in the code samples in this chapter. For information about compositional dependencies and the assetset tags, see "The ASSETSET (assetset) Tag Family" on page 586.

## Searchstates

How do you obtain the IDs of the flex assets that you want to display? With searchstate objects.

A **searchstate** is a set of search constraints based on the attribute values held in the `_Mungo` table for the flex asset type. You **apply** searchstates to **assetsets**.

You build a searchstate by adding or removing constraints to narrow or broaden the list of flex assets that are described by the searchstate. For example, the GE Lighting sample site uses searchstates to create drill-down searching features that visitors use to browse through the product catalog.

An unconstrained searchstate applied to an assetset creates an unfiltered list of all the assets of that type. For example, the following code sample would create an assetset that contains all the products in the GE Lighting catalog:

```
<SEARCHSTATE.CREATE NAME="nolimits"/>
<ASSETSET.SETSEARCHEDASSETS NAME="unconstrainedAssetSet"
   CONSTRAINT="nolimits" ASSETTYPES="Products"/>
```

To narrow the number of products in the assetset, you add constraints. For example, the following code sample would create an assetset that contains only the 40-watt light bulbs from the catalog:

```
<SEARCHSTATE.CREATE NAME="lightbulbs"/>
<SEARCHSTATE.ADDSIMPLESTANDARDCONSTRAINT NAME="lightbulbs"
    ATTRIBUTE="wattage" VALUE="40"/>
<ASSETSET.SETSEARCHEDASSETS NAME="40WattLightbulbs"
   CONSTRAINT="lightbulbs" ASSETTYPES="Products"/>
```

A constraint is a filter (restriction) that can be based on the value of an attribute or it can be based on another searchstate, which is called a nested searchstate.

A searchstate can search either the `_Mungo` table for the asset type database or the attribute indexes created by a search engine for that asset type. This means that you can mix database and rich-text (full-text through an index) searches in the same query. To apply a constraint against a search engine index, use the `SEARCHSTATE.ADDRICHTEXTCONSTRAINT` tag.

**Note**

Using SQL to query the flex asset database tables instead of using the `SEARCHSTATE` tag family is not supported.

The most commonly used `SEARCHSTATE` tags are as follows:

```
SEARCHSTATE.CREATE
SEARCHSTATE.ADDSTANDARDCONSTRAINT
SEARCHSTATE.ADDSIMPLESTANDARDCONSTRAINT
SEARCHSTATE.ADDRANGECONSTRAINT
SEARCHSTATE.ADDRICHTEXTCONSTRAINT
SEARCHSTATE.TOSTRING
SEARCHSTATE.FROMSTRING
```

All of the `SEARCHSTATE` tags are described in the *Content Server Tag Reference* and several of them are used in the code samples in this chapter.

## Assetsets, Searchstates, and Flex Attribute Asset Types

Because searchstates filter select assets based on attribute values, and assetsets are created by applying searchstates to the assets in the database, only those flex asset types that share the same attribute asset type can be included in the same assetset.

For example, in the GE Lighting sample site, you can create an assetset with both flex articles and flex images in it because they use the same attribute asset type—content asset attribute. However, because flex articles use the content attribute asset type and products use the product attribute asset type, you cannot create an assetset that contains both flex articles and product assets.

## Scope

The scope of assetsets and searchstates is local; that is, they exist only for the current element (rendered page).

When you want to maintain the existing searchstate, you can use the `SEARCHSTATE.TOSTRING` tag to convert it to a string and then include that string as an argument in the URL for the next page.

For example:

```
<SEARCHSTATE.TOSTRING NAME="ss" VARNAME="stringss"/>
<RENDER.SATELLITEPAGE
      pagename= SiteName/Products/Example
      ARGS_search="Variables.stringss"/>
```

And then, in the root element of this example page that receives the string, you code another searchstate:

```
<SEARCHSTATE CREATE NAME="ss"/>
```

And upack the string that was passed to the example element with a `SEARCHSTATE.FROMSTRING` tag:

```
<SEARCHSTATE.FROMSTRING NAME="ss" VALUE= "Variables.search"/>
```

# Coding Templates That Display Flex Assets

When you code templates for an online site that uses the flex asset model, you are primarily concerned with the values of flex attributes, which are assets themselves.

A flex asset (a product, for example) or flex parent asset considered in the context of displaying it, is really an abstraction of attribute values.

You use searchstates to obtain the identity of the flex assets that you want to display, filtering the assets under consideration by their attribute values. The result is an assetset of flex assets or flex parent assets that is based on attribute values, and you can then display the attribute values for the assets in the assetset.

Be sure that you understand the data model of the flex family (or families) that you are using before you begin coding template elements for your flex assets. For more information, read Chapter 11, "Data Design: The Asset Models" and Chapter 16, "Designing Flex Asset Types."

## Example Data Set for the Examples in This Section

The GE Lighting sample site and the Engage extensions to the Burlington Financial sample site illustrate the full power of the flex asset data model and the coding toolset delivered with CS-Direct Advantage. The templates and elements in the sample sites illustrate the code for fully functioning online sites that display a nearly real-world amount of data.

The code examples in this chapter of the *Content Server Developer's Guide* are much simpler than the elements in the sample sites. These examples start with simple assetsets and searchstates ("hello assetset" and "hello searchstate") that interact with a small, example data set.

The example data set used in these examples is based on the product flex family, as follows:

| Flex Asset Type | External Name (as displayed in Content Server interface) | Internal Name (as used in the Content Server database)* |
|---|---|---|
| flex attribute | product attribute | `PAttributes` |
| flex asset | product | `Products` |
| flex parent | product parent | `ProductGroups` |

* Always use the internal name of the asset type when you use the `ASSETTYPES` parameter for an `ASSETSET` tag.

The example products in this example data set are pairs of blue jeans that have the following attributes:

| Attribute | Data Type | Number of Values |
|---|---|---|
| sku | string | single |
| color | string | multiple |
| price | integer | single |
| style | text | single |

There are four pairs of blue jeans, defined as follows:

| sku | color | price | style |
|---|---|---|---|
| jeans-1 | blue | 35 | wide |
| jeans-2 | blue,black | 30 | straight |
| jeans-3 | black,green | 25 | straight |
| jeans-4 | green | 20 | wide |

## Examples of Assetsets with One Product (Flex Asset)

The code samples in this section do the following:

- Create an assetset that contains one pair of jeans, identified by its `sku` number
- Log a dependency between the product asset and the rendered page(let)
- Get and display the value for the `price` attribute and display it
- Get and display the values for the `color` attribute and display them
- Get and display the values for both the `price` and `color` attribute with the same tag (`ASSETSET.GETMULTIPLEVALUES`)

### Create a Searchstate and Apply It to an Assetset

This line of code creates an unfiltered searchstate named `ss`:

```
<SEARCHSTATE.CREATE NAME="ss"/>
```

Next, we can narrow the unfiltered searchstate named `ss` so that it finds a specific product in the sample data set, by providing the `sku` of the product:

```
<SEARCHSTATE.ADDSIMPLESTANDARDCONSTRAINT NAME="ss"
TYPENAME="PAttributes" ATTRIBUTE="sku" VALUE="jeans-2"/>
```

Now we can create an assetset named `as`, applying the searchstate named `ss` to it:

```
<ASSETSET.SETSEARCHEDASSETS NAME="as"
ASSETTYPES="Products"CONSTRAINT="ss" FIXEDLIST="true"/>
```

Since the value of the `sku` attribute is unique for each product asset, there is only one product in the assetset: the one whose `sku` value is `jeans-2`.

Because this searchstate was created by querying for a hard-coded attribute value—a sku value of “jeans-2”—we know the exact contents of the assetset. That is why we set the `FIXEDLIST` parameter to “true.” Now the `ASSETSET.SETSEARCHEDASSET` tag logs a compositional dependency for the product asset.

### Get the Price of the Product

Next, let’s extract the price of this pair of jeans:

```
<ASSETSET.GETATTRIBUTEVALUES NAME="as" ATTRIBUTE="price"
TYPENAME="PAttributes" LISTVARNAME="pricelist"/>
```

Notice that even though `price` is a single-value attribute (which means the product only has one price), the `ASSETSET.GETATTRIBUTEVALUES` tag returns the value of the `price` attribute as a list variable (`LISTVARNAME=“pricelist”`).

### Display the Price of the Product

Now the following line of code can display the price of the `jeans-2` product:

```
Price: <ics.listget listname=”pricelist” fieldname=”value”/>
And this is the result:
   Price: 30
```

### Get the Colors for the Product

Next, let’s determine which colors this pair of jeans is available in.

As specified above, the `color` attribute is a multiple-value attribute. Because the `ASSETSET.GETATTRIBUTEVALUES` tag works the same whether an attribute is a single-value or a multiple-value attribute, we use the tag exactly as we did for single-value `price` attribute:

```
<ASSETSET.GETATTRIBUTEVALUES NAME="as" ATTRIBUTE="color"
TYPENAME="PAttributes" LISTVARNAME="colorlist"/>
```

### Display the Colors of the Product

Now the following code can display the colors for the `jeans-2` product, and, because this product can have more than one color, the code loops through the list:

```
Colors: <LOOP LIST="colorlist">
<ics.listget listname=”colorlist” fieldname=”value”/>
 
         </LOOP>
```

And this is the result:

```
   Colors: black blue
```

### Create a List Object for the ASSETSET.GETMULTIPLEVALUES tag

In general, you should not use the `ASSETSET.GETATTRIBUTEVALUES` tag when you want to get the value for more than one attribute.

The `ASSETSET.GETMULTIPLEVALUES` tag gets and scatters the values from more than one attribute, for all the assets in an assetset. Because the tag makes only one call to the database for all the attribute values, it performs the query more efficiently than using multiple `ASSETSET.GETATTRIBUTEVALUES` tags.

Before you can use this tag, however, you must use the `LISTOBJECT` tags to create a list object that describes the attributes that the `ASSETSET.GETMULTIPLEVALUES` tag will return. The list object needs one row for each attribute that you want to get.

This next line of code creates a list object named `lo` that has columns named `attributetypename`, `attributename`, and `direction`.

```
<LISTOBJECT.CREATE NAME="lo"
COLUMNS="attributetypename,attributename,direction"/>
```

Then, this line adds a row to the list object for each attribute, `color` and `price`:

```
<LISTOBJECT.ADDROW NAME="lo" attributetypename="PAttributes"
attributename="color" direction="none"/>
<LISTOBJECT.ADDROW NAME="lo" attributetypename="PAttributes"
attributename="price" direction="none"/>
```

The next line of code converts the list object to a list variable name `lolist`:

```
<LISTOBJECT.TOLIST NAME="lo" LISTVARNAME="lolist"/>
```

### Get the Value for Both Price and Color with ASSETSET.GETMULTIPLEVALUES

And now we can get the values for both the price and the color attribute from our original assetset, named `as`:

```
<ASSETSET.GETMULTIPLEVALUES NAME="as" PREFIX="multi"
LIST="lolist" BYASSET="false"/>
```

### Display the Value of Price and Color for the jeans-2 Product

Now that the values are stored in the list variable (`lolist`), the following code can display all the values for all the attributes:

```
<LOOP LIST="lolist">
      <ics.listget listname="lolist" fieldname="attributename"
         output="attrName"/>
      <ics.getvar name="attrName"/> is
      <LOOP LIST="multi:Variables.attrName">
         <ics.listget listname="multi:Variables.attrName"
            fieldname="value"/> 
      </LOOP><P/>
</LOOP>
```

This code sets up a nested loop that loops through all the attributes in the `lolist` variable, and then loops through all the distinct attribute values for each of the attributes in the `lolist` list variable.

And this is the result:

```
color is blue black
price is 30
```

## Special Cases: Flex Attributes of Type Text, Blob, and URL

If you want to display the values held in flex attributes of type `text`, `blob`, or `url` (which was deprecated in version 4.0), use the methodologies described in this section.

### Flex Attributes of Type Text

The `ASSETSET.GETMULTIPLEVALUES` tag does not retrieve the values for attributes of type `text`. This means that you must include a separate `ASSETSET.GETATTRIBUTEVALUES` tag for attributes of this type.

For example, if the color attribute in the sample data set used in these examples were an attribute of type `text` rather than type `string`, we could not have retrieved its values with the `ASSETSET.GETMULTIPLEVALUES` tag in the preceding examples.

## Flex Attributes of Type Blob

The `blob` attribute type was new in version 4.0 and it replaced the attribute of type `url`.

As previously mentioned in Chapter 11, "Data Design: The Asset Models," the `_Mungo` table for a flex asset type stores the attribute values for the flex assets of that type and the `ASSETSET` tags query the asset type's `_Mungo` table for attribute values.

Attributes of type `blob` are an exception:

- CS-Direct Advantage stores all the values of all the attributes of type `blob` in the `MungoBlobs` table.
- A row in the `_Mungo` table (`Products_Mungo`, for example) for an attribute of type `blob` stores only the ID of the row in the `MungoBlobs` table that holds its value. That is, the `blob` column in a `_Mungo` table is a foreign key to the `MungoBlobs` table.

This means that for an attribute of type `blob`, the `ASSETSET.GETATTRIBUTEVALUES` and `ASSETSET.GETMULTIPLEVALUES` tags return the ID of the blob attribute's value, but not the actual value.

Once the ID of the attribute's value has been identified, you can do one of two things with it:

- Use the ID to obtain a BlobServer URL.
- Use the ID to extract the actual value of the blob.

### Creating a BlobServer URL

To obtain a BlobServer URL for the value of the flex attribute blob, you do the following:

- Use the `BLOBSERVICE` tags to programmatically identify the `MungoBlobs` table and the appropriate columns in it.
- Pass that information to a `RENDER.SATELLITEBLOB` tag, if you want the URL in an HTML tag, or to a `RENDER.GETBLOBURL` tag if you need only the URL without the HTML tag.

**Note**

Be sure to use the `BLOBSERVICE` tags to programmatically identify the `MungoBlobs` table, as shown in the following example. By obtaining the value with the `BLOBSERVICE` tags rather than hard coding the name of the table into your code, your code will function properly even if the table name is changed in a future version of the product.

To illustrate the following blob examples, let's add the following attribute to the jeans products in our sample data set:

| Attribute | Data Type | Number of Values |
|---|---|---|
| `description` | `blob` | `single` |

First, let's create the assetset and log the dependency between the `jeans-2` product and the rendered page:

```
<SEARCHSTATE.CREATE NAME="ss"/>
<SEARCHSTATE.ADDSIMPLESTANDARDCONSTRAINT NAME="ss"
TYPENAME="PAttributes" ATTRIBUTE="sku" VALUE="jeans-2"/>
<ASSETSET.SETSEARCHEDASSETS NAME="as" ASSETTYPES="Products"
CONSTRAINT="ss"/>
<ASSETSET.GETASSETLIST NAME="as" LISTVARNAME="aslist"/>
<RENDER.LOGDEP cid="aslist.assetid" c="aslist.assettype"/>
```

The next line of code gets the ID of the `jeans-2` asset's `description` attribute (that attribute of type `blob`) and stores it in a list variable called `descFile`

```
<ASSETSET.GETATTRIBUTEVALUES NAME="as" TYPENAME="PAttributes"
ATTRIBUTE="description" LISTVARNAME="descFile"/>
```

The next lines of code use the `BLOBSERVICE` tags to obtain the table name and column names from the CS-Direct Advantage table that stores the attribute values for blob attributes and store them in variables named "`uTabname`", "`idColumn`", and "`uColumn`":

```
<BLOBSERVICE.GETTABLENAME VARNAME="uTabname"/>
<BLOBSERVICE.GETIDCOLUMN VARNAME="idColumn"/>
<BLOBSERVICE.GETURLCOLUMN VARNAME="uColumn"/>
```

Now we can pass the list variable named `descFile` and the `uTabname`, `idColumn`, and `uColumn` variables to a `RENDER.SATELLITEBLOB` tag, which returns a BlobServer URL in an HTML tag:

```
<RENDER.SATELLITEBLOB
   BLOBTABLE="Variables.uTabname"
   BLOBWHERE="descFile.value"
   BLOBKEY="Variables.idColumn"
   BLOBCOL="Variables.uColumn"
   BLOBHEADER="application/pdf"
   /> add service="a href" ... download link...
```

The `RENDER.SATELLITEBLOB` tag returns a BlobServer URL in an HREF tag.

### Getting and Displaying the Value of the Blob

To obtain and display the contents or data in the flex attribute blob after its ID has been returned, you use a `BLOBSERVICE.READDATA` tag, which loads the file name and URL data of the blob.

Under the same assumptions about the data set that we used for the preceding blob example, let's create the assetset, log the dependency between the `jeans-2` asset and the rendered page, and get the ID of the `description` attribute's value:

```
<SEARCHSTATE.CREATE NAME="ss"/>
<SEARCHSTATE.ADDSIMPLESTANDARDCONSTRAINT NAME="ss"
TYPENAME="PAttributes" ATTRIBUTE="sku" VALUE="jeans-2"/>
<ASSETSET.SETSEARCHEDASSETS NAME="as" ASSETTYPES="Products"
CONSTRAINT="ss"/>
<ASSETSET.GETASSETLIST NAME="as" LISTVARNAME="aslist"/>
<RENDER.LOGDEP cid="aslist.assetid" c="aslist.assettype"/>
<ASSETSET.GETATTRIBUTEVALUES NAME="as" TYPENAME="PAttributes"
ATTRIBUTE="description" LISTVARNAME="descFile"/>
```

This time, we want to get and then display the value (data) of the description attribute, so we have to use the BLOBSERVICE.READDATA tag:

```
<BLOBSERVICE.READDATA ID="descFile.value" LISTVARNAME="descData"/>
<ics.listget listname="descData" fieldname="@urldata"/>
```

## Flex Attributes of Type URL

Attributes of type url were deprecated in the 4.0 version of the product. You should use attributes of type blob, instead.

However, if you have upgraded from a version 3.6.3 and you have attributes of type url whose values you want to display, be sure that you complete these extra coding steps:

- Obtain the value of the property that sets the defdir for the URL columns in the CS-Direct Advantage tables and store it in a variable.
- Use the variable with an INSERT tag rather than a CSVAR tag.

This time, let's assume that the ID of the asset has been passed to this element in a cid variable and that the description attribute is of type url rather than of type blob.

When the cid variable is set, you can create the assetset like this:

```
<ASSETSET.SETASSET NAME="as" TYPE="Products"
ID="Variables.cid"/>
```

The next line of code obtains the value of the description attribute (which is of type url in this example):

```
<ASSETSET.GETATTRIBUTEVALUES NAME="as" ATTRIBUTE="urlattr"
LISTVARNAME="attr_list"/>
```

Now we need the value of the cc.urlattrpath property from the gator.ini file:

```
<PROPERTY.GET INIFILE="gator.ini" PARAM="cc.urlattrpath"
VARNAME="path"/>
```

And finally, we use the INSERT tag rather than the CSVAR tag to display the value of the description attribute:

```
<INSERT URL="Variables.pathattr_list.value"/>
```

# Examples of Assetsets with More Than One Product (Flex Asset)

The code samples in this section do the following:

- Create an assetset that holds all the products (pairs of jeans) in the sample data set being used in this chapter.
- Get and display a count of the number of jeans in the assetset.
- Get and display all the values for the color attribute for all the pairs of jeans in the assetset.
- Get and display all the values for both the color and the style attributes for the jeans in the assetset.
- Get and display, in a table, all the attribute values for the jeans in the assetset.
- Add a search constraint that filters the assetset for the jeans whose price falls into a specific range.

- Replace the range constraint on the price attribute with a search constraint that filters the assetset for the jeans that are available in any color that begins with the letter “b.”
- Replace that color constraint with one that filters the assetset for the jeans that are available in either of two specific colors: blue or black

### Create a Searchstate and Apply it to an Assetset

This line of code creates an unfiltered searchstate named `ss`:

```
<SEARCHSTATE.CREATE NAME="ss"/>
```

When you apply the unfiltered searchstate to an assetset, you get all the flex assets of the type specified (in this case, product assets):

```
<ASSETSET.SETSEARCHEDASSETS NAME="as" CONSTRAINT="ss"
ASSETTYPES="Products"/>
```

### Display the Number of Assets in the Assetset

These lines of code return and display a count of the number of assets in the assetset, which at this point represents the entire sample catalog:

```
<ASSETSET.GETASSETCOUNT NAME="as" VARNAME="count"/>
How many products are in the catalog?
<ics.getvar name=”count”/>
```

And this is the result:

```
How many products are in the catalog? 4
```

### Display the Colors That the Jeans Are Available In

The next lines of code get and display the different colors for the jeans. In other words, the distinct values of the `color` attribute:

```
<ASSETSET.GETATTRIBUTEVALUES NAME="as" ATTRIBUTE="color"
TYPENAME="PAttributes" LISTVARNAME="colors"/>

What are the possible colors for any pair of jeans?<BR/>

<LOOP LIST="colors">
      <ics.listget listname=”colors” fieldname=”value”/>
</LOOP><p/>
```

And this is the result:

```
What are the possible colors for any pair of jeans?
black blue green
```

### Display Both the Colors and the Styles for the Jeans in the Assetset

Next, let’s extract and display the values for both the `color` and the `style` attribute for the jeans in the assetset. This time we use the `ASSETSET.GETMULTIPLEVALUES` tag.

First, however, we need to create a list object for the resultset that the `ASSETSET.GETMULTIPLEVALUES` tag returns. The list object needs one row for each of the attributes, as follows:

```
<LISTOBJECT.CREATE NAME="lo"
COLUMNS="attributename,attributetypename,direction"/>
<LISTOBJECT.ADDROW NAME="lo" attributename="color"
attributetypename="PAttributes" direction="none"/>
<LISTOBJECT.ADDROW NAME="lo" attributename="style"
attributetypename="PAttributes" direction="none"/>
```

The next line of code converts the list object to a list variable named `lolist`:

```
<LISTOBJECT.TOLIST NAME="lo" LISTVARNAME="lolist"/>
```

Now we can extract the attributes and store them in the list variable named `lolist`:

```
<ASSETSET.GETMULTIPLEVALUES NAME="as" LIST="lolist"
PREFIX="distinct" BYASSET="false"/>
```

Notice the `BYASSET` parameter in the preceding line of code. Because there is more than one asset in the assetset and we want to know the distinct values for the attribute rather than all the attribute values for each asset in the assetset, `BYASSET="false"`. This way, we get only the unique attribute values and not every single attribute value.

The next lines of code loop through the list and display the unique values for each attribute:

```
Here are all the possible colors:
<LOOP LIST="distinct:color">
      <ics.listget listname="distinct:color" fieldname="value"/>
</LOOP><P/>

Here are all the possible styles:
<LOOP LIST="distinct:style">
      <ics.listget listname="distinct:style" fieldname="value"/>
</LOOP><P/>
```

And this is the result:

```
Here are all the possible colors: green blue black
Here are all the possible styles: wide straight
```

### Create a Table That Displays All the Jeans and Their Attribute Values

You can also use the `ASSETSET.GETMULTIPLEVALUES` tag to obtain the attribute values that are distinct for each asset in the assetset. It creates a list of all the products and the values for their attributes that we can use to create a grid or table that displays all the products in the example catalog.

In this case, we have to do two additional things:

- Because we want the attribute values grouped by the asset that they belong to, the `BYASSET` parameter must be set to `"true"`.
- Because we need the IDs of the assets in this case, we need to use the `ASSETSET.GETASSETLIST` tag to obtain them.

First, this code creates a list object:

```
<LISTOBJECT.CREATE NAME="lo"
COLUMNS="attributename,attributetypename,direction"/>
<LISTOBJECT.ADDROW NAME="lo" attributename="color"
attributetypename="PAttributes" direction="none"/>
<LISTOBJECT.ADDROW NAME="lo" attributename="style"
attributetypename="PAttributes" direction="none"/>
<LISTOBJECT.ADDROW NAME="lo" attributename="price"
attributetypename="PAttributes" direction="none"/>
<LISTOBJECT.ADDROW NAME="lo" attributename="sku"
attributetypename="PAttributes" direction="none"/>
<LISTOBJECT.TOLIST NAME="lo" LISTVARNAME="lolist"/>
```

Next, we can get the attribute values:

```
<ASSETSET.GETMULTIPLEVALUES NAME="as" LIST="lolist"
PREFIX="grid" BYASSET="true"/>
```

And then we use the `ASSETSET.GETASSETLIST` tag.

```
<ASSETSET.GETASSETLIST NAME="as" LISTVARNAME="aslist"/>
```

It returns a list with these columns:

- assettype
- assetid

By using both lists, we can create a grid that shows all of the products and all of their attribute values:

```
<TABLE>
<LOOP LIST="aslist">
      <TR>
         <TD><CSVAR NAME="grid:aslist.assetid:sku.value"/></TD>
         <TD><CSVAR NAME="grid:aslist.assetid:price.value"/>
            </TD>
         <TD><CSVAR NAME="grid:aslist.assetid:style.value"/>
            </TD>
         <TD>
            <IF
COND="IsList.grid:aslist.assetid:color=true"><THEN>
            <LOOP LIST="grid:aslist.assetid:color">
            <CSVAR NAME="grid:aslist.assetid:color.value"/
> 
            </LOOP>
            </THEN></IF>
         </TD>
      </TR>
</LOOP>
</TABLE>
```

And this is the result:

| | | | |
|---|---|---|---|
| jeans-1 | 35 | wide | blue |
| jeans-2 | 30 | straight | black blue |
| jeans-3 | 25 | straight | black green |
| jeans-4 | 20 | wide | green |

### Search for Jeans Based on a Range of Prices

Up until now, we have been using the same assetset (`NAME="as"`) that was created in the second line of code in this section. Next, let's filter the assetset by the price attribute, using a range constraint.

This line of code adds a range constraint to our original searchstate (`NAME="ss"`) that was created in the first line of code in this section:

```
<SEARCHSTATE.ADDRANGECONSTRAINT NAME="ss" ATTRIBUTE="price"
TYPENAME="PAttributes" LOWER="0" UPPEREQUAL="30"/>
```

The range is from 0 to 30. Let's apply the modified searchstate against our assetset:

```
<ASSETSET.SETSEARCHEDASSETS NAME="as" CONSTRAINT="ss"
ASSETTYPES="Products"/>
```

And check whether it worked, by obtaining and displaying a count of the jeans that are now in the assetset:

```
<ASSETSET.GETASSETCOUNT NAME="as" VARNAME="count"/>
How many jeans products are less than or equal to $30?
<ics.getvar name="count"/>
```

Here's the result:

```
How many jeans products are less than or equal to $30? 3
```

### Search for Jeans with a Wildcard for Color

Now let's replace the range constraint on the `price` attribute with a search constraint that filters the assetset for the jeans that are available in any color that begins with the letter "b."

First this line of code deletes the range constraint for price:

```
<SEARCHSTATE.DELETECONSTRAINT NAME="ss" ATTRIBUTE="price"/>
```

And this line of code adds a new constraint for color, using the percentage (%) character as a wildcard with the VALUE parameter:

```
<SEARCHSTATE.ADDSIMPLELIKECONSTRAINT NAME="ss"
ATTRIBUTE="color" TYPENAME="PAttributes" VALUE="b%"/>
```

The `VALUE="b%"` statement means "any color that begins with the letter "b." Lets apply the modified searchstate against our same assetset (`as`):

```
<ASSETSET.SETSEARCHEDASSETS NAME="as" CONSTRAINT="ss"
ASSETTYPES="Products"/>
```

And check whether it worked by obtaining and displaying a count of the number of jeans that are in the assetset now:

```
<ASSETSET.GETASSETCOUNT NAME="as" VARNAME="count"/>
How many jeans have a color that begins with the letter "b"?
<ics.getvar name="count"/>
```

Here's the result:

```
How many jeans have a color that begins with the letter "b"? 3
```

### Search for Jeans with Specific Colors

Finally, let's change the color constraint that filters the assetset for the jeans that are available in either of two specific colors: blue or black

This line of code deletes the color constraint from the searchstate:

```
<SEARCHSTATE.DELETECONSTRAINT NAME="ss" ATTRIBUTE="color"/>
```

Next, because we want to filter based on two values for the color attribute, we need to create a list object with those values:

```
<LISTOBJECT.CREATE NAME="lo" COLUMNS="value"/>
<LISTOBJECT.ADDROW NAME="lo" value="blue"/>
<LISTOBJECT.ADDROW NAME="lo" value="black"/>
<LISTOBJECT.TOLIST NAME="lo" LISTVARNAME="colorlist"/>
```

Now we can use the list variable named `colorlist` to create the searchstate:

```
<SEARCHSTATE.ADDSTANDARDCONSTRAINT NAME="ss" ATTRIBUTE="color"
TYPENAME="PAttributes" LIST="colorlist"/>
```

The `LIST="colorlist"` statement is the equivalent of the `VALUE` statement in the preceding example. It means "attribute values that match any of the colors in the list named `colorlist`". Let's apply the modified searchstate to our same assetset:

```
<ASSETSET.SETSEARCHEDASSETS NAME="as" CONSTRAINT="ss"
ASSETTYPES="Products"/>
```

And check whether it worked by obtaining and displaying a count of the number of jeans that are in the assetset now:

```
<ASSETSET.GETASSETCOUNT NAME="as" VARNAME="count"/>
How many products have a color that is black or blue?
<ics.getvar name="count"/>
```

Here's the result:

```
How many products have a color that is black or blue? 3
```

# Creating URLs for Hyperlinks

Whether your site is dynamic or static, the fact that you are using a Content Server system indicates that your content changes regularly. That means that you cannot hard code URLs into hyperlinks. Your pages must be able to determine the identity of the assets they are providing links to when the page is rendered, either by the Export to Disk publish method or by a visitor's browser on a dynamic site.

CS-Direct provides three tags (each with an XML and a JSP version) that you can use to create your URLs:

- For URLs for assets that are not blobs, use `RENDER.GETPAGEURL` tag.
- For URLs for assets that are blobs, use either the `RENDER.SATELLITEBLOB` tag or the `RENDER.GETBLOBURL` tag.

## RENDER.GETPAGEURL (render:getpageurl)

To obtain URLs for regular assets (that is, assets that are not blobs), use the `RENDER.GETPAGEURL` tag.

The `RENDER.GETPAGEURL` tag processes arguments passed in from the element that invokes it into a URL-encoded string that it returns as a variable that you name with the `OUTSTR` parameter. By convention, the name of that variable is `referURL`.

If rendermode is set to `export`, it creates a static URL (unless you specify that it should be dynamic). If rendermode is set to `live`, it creates a dynamic URL.

For example:

```
<RENDER.GETPAGEURL PAGENAME="BurlingtonFinancial/Article/Full
      cid="Variables.cid"
      c="Article"
      p="Variables.p"
      OUTSTR="referURL"/>
```

You can now use the value in the referURL variable to create a hyperlink with an <A HREF> tag.

For more information about this tag, see the *Content Server Tag Reference*.

## RENDER.SATELLITEBLOB (render:satelliteblob)

Binary large objects (blobs) that are stored in the Content Server database are served by the BlobServer servlet rather than the ContentServer servlet. The `RENDER.SATELLITEBLOB` tag returns an HTML tag with a BlobServer URL in it.

This tag takes a set of arguments that define the blob and an additional set of arguments that determine how to format the blob. For example, you can use it to create an `<IMG SRC>` tag or an `<A HREF>` tag, as follows:

```
<RENDER.SATELLITEBLOB
   BLOBTABLE="ImageFile"
   BLOBKEY="id"
   BLOBCOL="urlpicture"
   BLOBWHERE="Variables.asset:id"
   BLOBHEADER="Variables.asset:mimetype"
   SERVICE="IMG SRC"
   ARGS_alt="Variables.asset:alttext"
   ARGS_hspace="5" ARGS_vspace="5"/>
```

Note that there are additional coding steps if you are creating a URL for a flex attribute of type `blob`. For information, see "Flex Attributes of Type Blob" on page 606.

For a longer example, examine the Burlington Financial imagefile template named TeaserSummary. You can examine it in two ways:

- Search for and then inspect it in the Content Server interface.
- Use Content Server Explorer to open the template element called:

  `ElementCatalog/BurlingtonFinancial/ImageFile/TeaserSummary.`

Even if you are not using Satellite Server, you should still use the `RENDER.SATELLITEBLOB` tag because the tag can create a BlobServer URL in an HTML tag even when Satellite Server is not present.

For more information about this tag, see the *Content Server Tag Reference*.

## RENDER.GETBLOBURL (render:getbloburl)

If you need a BlobServer URL only, without it being embedded in an HTML tag, use the `RENDER.GETBLOBURL` tag.

For example, the Burlington Financial element named `SetHTMLHeader` uses the `RENDER.GETBLOBURL` element to obtain a BlobServer URL (stored as a variable named `referURL`) that it then passes on to JavaScript code that runs on the client side to determine which browser the visitor is using. In this case, the client-side JavaScript creates the HTML tag based on the browser it discovers, so it needs the BlobServer URL without an HTML tag.

The `SetHTMLHeader` element is the element for a CSElement. You can examine it in two ways:

- Use the Content Server interface to search for the BurlingtonFinanical/Common/SetHTMLHeader CSElement and then inspect it.

- Use Content Server Explorer to open the element called:

  `ElementCatalog/BurlingtonFinancial/Common/SetHTMLHeader`

Note that there are additional coding steps if you are creating a URL for a flex attribute of type `blob`. For information, see "Flex Attributes of Type Blob" on page 606.

For more information about the `RENDER.GETBLOBURL` tag, see the *Content Server Tag Reference*.

## Using the referURL Variable

The `RENDER.GETPAGEURL`, `RENDER.GETBLOBURL`, and `RENDER.SATELLITEBLOB` tags were introduced in the 3.6.x version of CS-Direct. Older versions of the product used elements named `GetPageURL` and `GetBlobURL` to obtain URLs; they are coded to return URLs in a variable named `referURL`.

By convention, all of the sample code in the sample sites that use the tags that replaced the `GetPageURL` and `GetBlobURL` elements use a `referURL` variable for the value of the URL.

Do not append or add any text to the value held in the `referURL` variable or any other variable returned by a `RENDER.GETPAGEURL` or `RENDER.GETBLOBURL` tag. URLs in this kind of variable are complete (whole). If you change the URL returned by the tag, you are likely to break it.

If you need to include additional arguments in a URL, use the `RENDER.PACKARGS` tag to URL-encode them ("pack" them) and then pass those encoded arguments to the `RENDER.GETPAGEURL` or `RENDER.GETBLOBURL` tag with the `PACKEDARGS` parameter.

For information about the `RENDER.PACKARGS` tag, see the *Content Server Developer's Guide*.

# Handling Error Conditions

While you code your elements, you should also include code that checks for error conditions. You decide which error conditions are serious and, when necessary, code a solution or alternate action. Sometimes the solution is to write a meaningful error message. As an additional step, you can include code that stops a broken page from being cached.

**Note**

While debugging your code, you can use the `commons-logging.properties` and `log4j.properties` files to enable loggers. Error and debugging messages are then written to Content Server's log file (named `futuretense.txt`, by default), which is typically located in the Content Server installation directory.

For information about the debugging properties, see the *Content Server Property Files Reference*.

## Using the Errno Variable

The `errno` variable, a standard Content Server variable, holds error numbers that the Content Server XML and JSP tags report. When a Content Server tag cannot successfully execute, it sets errno to the value that best describes the reason why it did not succeed. For example, an errno value of -13004 means a `CURRENCY` tag couldn't read a number because it was not in the correct currency format. For a complete list of all the errno values and their descriptions, see the error conditions section in the *Content Server Tag Reference*.

The tags that are delivered with the CS modules and products clear errno before they execute so you do not need to set errno to 0 when you want to check for errors from these tags. Here's a code example that determines whether an `ASSET.LOAD` was successful before attempting to load the child assets:

```
<ASSET.LOAD NAME="topArticle" TYPE="Article"
OBJECTID="Variables.cid"/>
      <IF COND="IsError.Variables.errno=false">
         <THEN>
            <ASSET.CHILDREN NAME="topArticle"
               LIST="listOfChildren”/>
         </THEN>
      </IF>
```

If you want to check the results of the tags that are delivered by Content Server, you must include code that clears the value errno before the tag whose results you want to check. For example:

```
<SETVAR NAME=“errno” VALUE=“0”/>
```

For a longer example, see the Burlington Financial CSElement named BurlingtonFinancial/Util/Account/SignUp. This CSElement provides the code that adds members to the site and updates existing member's information. It checks for several error conditions and provides appropriate responses to them.

The following code sample shows an error message that you could use while you are in the process of developing your templates:

```
<IF rendermode=“preview”>
```

```
        <THEN>
           <IF COND="IsError.Variables.errno=true">
              <THEN>
              <FONT COLOR="#FF0000">
              Error <ics.geterrno/>
              while rendering <ics.getvar name="pagename"/>
              with asset ID <ics.getvar name="cid"/>.
              </FONT>
              </THEN>
           </IF>
        </THEN>
</IF>
```

## Ensuring that Incorrect Pages Are Not Cached

If you can determine that the output from an element is incorrect, there is probably no need for Content Server or Satellite Server to cache the page. You can stop the page that is being generated from being cached with the `ics.disablecache` tag.

### Example 1: Error Condition

To continue with the first example in "Using the Errno Variable" on page 616, if the article asset could not be loaded, there would also be no reason to cache the page. You could add the following `ELSE` statement to the `IF` condition in that code sample:

```
<ASSET.LOAD NAME="topArticle" TYPE="Article"
OBJECTID="Variables.cid"/>
      <IF COND="IsError.Variables.errno=false">
         <THEN>
            <ASSET.CHILDREN NAME="topArticle"
               LIST="listOfChildren"/>
         </THEN>
      <ELSE>
         <ics.disablecache/>
      </ELSE>
      </IF>
```

### Example 2: Clear the Page From Cache if the Asset's Status is VO (Basic Assets Only)

As described in "CacheManager and Dynamic Publish Sessions" on page 585, the CacheManager on the destination system regenerates all the pages and pagelets that were affected by a publishing session. "Affected pages" includes those whose dependent assets were deleted.

Deleted assets have their status set to `VO`. The `ASSET.LOAD` and `asset:load` tags do not check the status of an asset before they execute which means they can and will load a deleted asset. Typically this isn't a problem. Why? Because an asset cannot be deleted until all links to it from other assets are removed. Therefore, when the site is regenerated there are no longer any links to a page or pagelet that would display the deleted asset. But there is no need to leave a page or pagelet that displays a deleted asset in the cache.

The following code sample stops the page from being cached if the asset cannot be loaded or if the asset's status is deleted:

```
<ASSET.LOAD TYPE="Variables.c" OBJECTID="Variables.cid"
NAME="WireStoryTextArticle"/>
<!-- if the asset cannot be loaded, then flush the pagelet from
cache -->
      <if COND="IsError.Variables.errno=true">
         <then>
         <ics.disablecache/>
         </then>
      </if>
<ASSET.SCATTER NAME="WireStoryTextArticle" PREFIX="asset"/>
<!-- if the asset is marked as void, then flush the pagelet
from cache -->
      <if COND="Variables.asset:status=VO">
         <then>
         <ics.disablecache/>
         </then>
      </if>
```

Note that you do not need to include code that checks the status of flex assets. The `SEARCHSTATE` and `searchstate` tags do not return assets that have a status of VO and the `ASSETSET` and `assetset` tags do not include assets that have a status of VO in the assetsets that they create.

Chapter 28

# Template Element Examples for Basic Assets

This chapter uses examples from the Burlington Financial sample site to illustrate the information presented in Chapter 27, "Coding Elements for Templates and CSElements." It contains the following sections:

- Example 1: Basic Modular Design
- Example 2: Coding Links to the Article Assets in a Collection Asset
- Example 3: Using the ct Variable
- Example 4: Coding Templates for Query Assets
- Example 5: Displaying an Article Asset Without a Template
- Example 6: Displaying Site Plan Information
- Example 7: Displaying Non-Asset Information

All of the elements described in this section are from the Burlington Financial sample site.

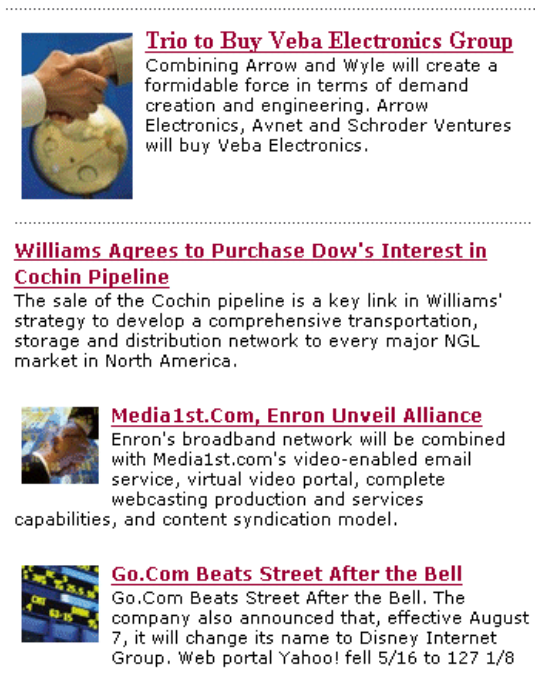
# Example 1: Basic Modular Design

The Burlington Financial sample site is an example of a modular site design that takes advantage of common elements so that common code is written once but reused in several locations or contexts. Following is a description of how one area on the Burlington Financial home page is created from five separate elements.

First, open the Content Server interface, select the Burlington Financial site and preview the Home page asset. You can either search for the asset and select **Preview** from the drop-down list on the icon bar or you can expand the **Placed Pages** icon under the **Burlington Financia**l node in the **Site Plan** tab, select the **Home** page, and then select **Preview** from the right-mouse menu.

CS-Direct displays the Burlington Financial home page in your browser.

Directly under the date, there is a column that displays the main stories of the day. There is a summary paragraph and byline for each story in the list. The titles of the stories are hyperlinks to the full story. Several of the stories, including the first story in the list, also present a photo:




This example describes how the first story in the list is identified, selected, positioned at the top of the list, and formatted.

These are the elements used to format the first story in the list:

- BurlingtonFinanical/Page/Home
- BurlingtonFinancial/Collection/MainStoryList
- BurlingtonFinancial/Article/LeadSummary
- BurlingtonFinancial/ImageFile/TeaserSummary

## First Element: Home

The Home page of the of the Burlington Financial sample site uses a template that is also named Home. You can examine it in two ways:

- Search for and then inspect it in the Content Server interface.
- Use Content Server Explorer to open the template element called:

  `ElementCatalog/BurlingtonFinancial/Page/Home`

First, the `Home` element loads the home page asset, names it HomePage, and then scatters the information in its fields:

```
<ASSET.LOAD TYPE="Variables.c" OBJECTID="Variables.cid"
NAME="HomePage"/>
<ASSET.SCATTER NAME="HomePage" PREFIX="asset"/>
```

The value for cid is passed in from the Burlington Financial URL and the value for c is available because it is set as a variable in the resarg1 column in the SiteCatalog page entry for the Home template.

Scroll down past several `callelement` and `RENDER.SATELLITEPAGE` tags to the following `ASSET.CHILDREN` tag:

```
<ASSET.CHILDREN NAME="HomePage" LIST="MainStories"
CODE="TopStories"/>
```

With this code, Home obtains the collection asset identified as the page asset's TopStories collection (`CODE="TopStories"`) and creates a list named MainStories to hold it (`LIST="MainStories"`).

Next, Home determines whether it successfully obtained the collection and then calls for the page entry of the MainStoryList template.

```
<IF COND = "IsList.MainStories=true">
<THEN>
<RENDER.SATELLITEPAGE pagename="BurlingtonFinanical/Collection/
MainStoryList"
ARGS_cid="MainStories.oid"
ARGS_p="Variables.asset:id"/>
<THEN/>
<IF/>
```

Notice that Home passes the identity of the list that holds the collection to MainStories with `ARGS_cid` and the identity of the Home page asset with `ARGS_p="Variables.asset:id"`.

## Second Element: MainStoryList

The MainStoryList page entry invokes its root element. Use Content Server Explorer to open and examine this element:

`ElementCatalog/BurlingtonFinancial/Collection/MainStoryList.xml`

The MainStoryList element is the template element (the root element) for the MainStoryList Template asset, a template that formats collection assets.

This element creates the framework for the Home page column that holds the main list of stories, and then fills that column with the articles from the TopStories collection. It uses two templates to format those articles:

- LeadSummary for the first article in the collection (the top-ranked article)
- Summary for the rest of the articles

Because the purpose of this example is to describe how the first story in the list is displayed, this example discusses only the LeadSummary template element.

MainStoryList loads and scatters the collection that Home passed to it:

```
<ASSET.LOAD TYPE="Variables.c" OBJECTID="Variables.cid"
      NAME="MainStoryListCollection"/>
<ASSET.SCATTER NAME="MainStoryListCollection" PREFIX="asset"/>
```

It then extracts the articles from the collection, creates a list to hold them, ordering them by their rank:

```
<ASSET.CHILDREN NAME="MainStoryListCollection"
LIST="theArticles"
ORDER="nrank" CODE="-"/>
```

And then it calls for the page entry of the LeadSummary template:

```
<RENDER.SATELLITEPAGE PAGENAME="BurlingtonFinancial/Article/
LeadSummary"
ARGS_cid="theArticles.oid"
ARGS_ct="Full"
ARGS_p="Variables.p"/>
```

Once again, this element passes on several pieces of information:

- The identity of the list that holds the articles (`ARGS_cid`)
- The name of the template to use when creating the link to each of the articles (`ARGS_ct`)
- The identity of the originating page asset (`ARGS_p`), which is Home.

Because the list was ordered by rank and this code does not loop through the list, the value in `ARGS_cid` (`theArticles.oid`) is the object ID of the highest ranked article in the collection because that article is the first article in the list.

## Third Element: LeadSummary

The LeadSummary page entry invokes its root element (which is the template element for the LeadSummary template). Use Content Server Explorer to open and examine it:

`ElementCatalog/BurlingtonFinancial/Article/LeadSummary.xml`

This element formats the first article in the TopStories collection. It does the following:

- Retrieves the image file associated with the first article through the TeaserImage association.
- Invokes the TeaserSummary element to obtain the formatting code for the image.
- Uses a `RENDER.GETPAGEURL` tag to obtain the URL for the first article in the collection.
- Displays the imagefile asset, the title of the article as a hyperlink to the full article, the summary paragraph, and the byline.

First LeadSummary loads the article and names it LeadSummaryArticle:

```
<ASSET.LOAD TYPE="Variables.c" OBJECTID="Variables.cid"
NAME="LeadSummaryArticle"/>
<ASSET.SCATTER NAME="LeadSummaryArticle" PREFIX="asset"/>
```

It obtains the assets associated with the article as its Teaser imagefile asset, creating a list for that file named “TeaserImage”:

```
<ASSET.CHILDREN NAME="LeadSummaryArticle" LIST="TeaserImage"
CODE="TeaserImageFile"/>
```

Finally, it calls the page entry for the TeaserSummary template, passing it the ID of the imagefile asset held in the list:

```
<THEN>
<RENDER.SATELLITEPAGE PAGENAME="BurlingtonFinancial/ImageFile/
TeaserSummary"
ARGS_cid="TeaserImage.oid"/>
</THEN>
</IF>
```

## Fourth Element: TeaserSummary

The TeaserSummary page entry invokes its root element, the template element for the TeaserSummary template. Use Content Server Explorer to open and examine it:

`ElementCatalog/BurlingtonFinancial/ImageFile/TeaserSummary`

Because imagefile assets are blobs stored in the Content Server database, and blobs stored in the database must be served by the BlobServer servlet rather than the ContentServer servlet, this element obtains an HTML tag that uses a BlobServer URL.

Scroll down to the following `RENDER.SATELLITEBLOB` tag:

```
<RENDER.SATELLITEBLOB BLOBTABLE=“ImageFile” BLOBKEY=“id”
BLOBCOL=“urlpicture” BLOBWHERE=“Variables.asset:id” BLOBHEADER=
“Variables.asset:mimetype” SERVICE=“IMG SRC” ARGS_alt=
“Variables.asset:alttext” ARGS_hspace=“5” ARGS_vspace=“5” />
```

The tag creates an HTML <IMG SRC> tag. The SRC is the blob in the ImageFile table identified through the ID passed in with BLOBWHERE="Variables.asset:id" and both its horizontal and vertical spacing are at five pixels.

When TeaserSummary is finished, LeadSummary continues.

## Back to LeadSummary

When LeadSummary resumes, having obtained the teaser image for the first article in the TopStories collection, it uses RENDER.GETPAGEURL to obtain the URL for that article:

```
<RENDER.GETPAGEURL PAGENAME="BurlingtonFinancial/Article/
Variables.ct"
      cid="Variables.cid"
      c="Article"
      p="Variables.p"
      OUTSTR="referURL"/>
```

Remember that when the MainStoryList element called the page entry for LeadSummary, it passed a ct variable set to Full. Therefore, the page name that LeadSummary is passing to RENDER.GETPAGEURL is really BurlingtonFinancial/Article/Full.

RENDER.GETPAGEURL creates the URL for the article based on the information passed in to it and then returns that URL to LeadSummary in a variable called referURL, as specified by the OUTSTR parameter.

LeadSummary uses the referURL variable in an HTML <A HREF> tag and then displays the link, the abstract of the article, and the byline:

```
<A class="featurehead" HREF="Variables.referURL"
REPLACEALL="Variables.referURL">
<csvar NAME="Variables.asset:description"/></A>
 <BR/>
<span class="thumbtext"><csvar NAME="Variables.asset:abstract"/
>
</span><BR/>
<span class="thumbcredit"><csvar NAME="Variables.asset:byline"/
>
</span><BR/>
```

Note the use of the REPLACEALL tag as an attribute in the HTML <A HREF> tag. You must use this tag as an attribute when you want to use XML variables in HTML tags.

Now that LeadSummary is finished, MainStoryList continues.

## Back to MainStoryList

Next MainStoryList loops through the rest of the articles in the TopStories collection and uses the Summary template to format them.

If you are interested, use Content Server Explorer to open and examine it:

ElementCatalog/Article/Summary

When MainStoryList is finished, Home continues.

## Back to Home

Home resumes, with a call to the WireFeedBox page entry.

# Example 2: Coding Links to the Article Assets in a Collection Asset

When an element needs URLs to create a list of hyperlinks to dynamically served Content Server pages, use the `RENDER.GETPAGEURL` tag.

These are the elements referred to in this example:

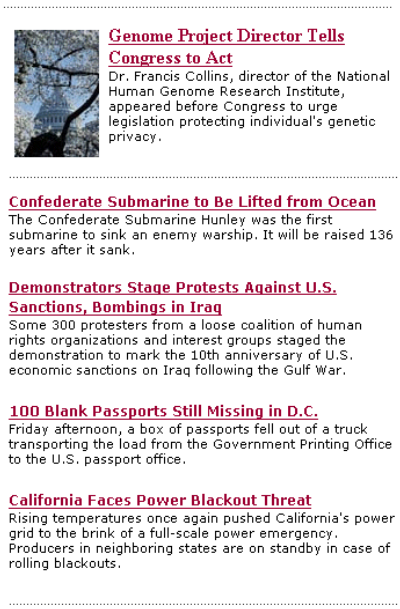
- ElementCatalog/BurlingtonFinancial/Page/SectionFront
- ElementCatalog/BurlingtonFinancial/Collection/PlainList

For the purposes of this example, the code displayed is stripped of any error checking so that you can focus on how the links are created.

## First element: SectionFront

SectionFront is the template element, the root element, of the SectionFront template which is assigned to the main section pages—News, Markets, Stocks, and so on. It is invoked when a visitor clicks a link to a section.

One section of a page formatted with the SectionFront element displays a list of links to articles from the Section Highlights collection that is associated with that page asset.




Use Content Server Explorer to open and examine the `SectionFront` element:

`ElementCatalog/BurlingtonFinancial/Page/SectionFront.`

First, `SectionFront` uses the variables `c` and `cid` to load and scatter the page asset, and names it “SectionFrontPage”:

```
<ASSET.LOAD TYPE="Variables.c" OBJECTID="Variables.cid"
NAME="SectionFrontPage"/>
<ASSET.SCATTER NAME="SectionFrontPage" PREFIX="asset"/>
```

The values for `c` and `cid` are passed to the `SectionFront` element from the link that invoked it. That link could be from the home page or any one of several other locations.

In Content Server Explorer, scroll down past several `ASSET.CHILDREN` tags to the one that retrieves the Section Highlights collection:

```
<ASSET.CHILDREN NAME="SectionFrontPage"
LIST="SectionHighlights" CODE="SectionHighlight"/>
```

This code retrieves the collection with the `CODE="SectionHighlights"` statement and stores it as a list, also named `SectionHighlights`.

Then SectionFront calls the page entry of the PlainList template (a collection template):

```
<RENDER.SATELLITEPAGE
pagename="BurlingtonFinancial/Collection/PlainList"
ARGS_cid="SectionHighlights.oid" ARGS_p="Variables.asset:id"/>
```

This code passes in the ID of the Section Highlights collection (`cid`) and the ID of the current page asset (`p`), which is the page asset assigned the name of SectionFrontPage.

## Second element: PlainList

The PlainList page entry invokes its root element, the template element for the PlainList template. Use Content Server Explorer to open and examine it:

`ElementCatalog/BurlingtonFinancial/Collection/PlainList.`

PlainList extracts the articles from the collection and presents them in a list, by their rank, with the subheadline of the article. This element assumes that the assets in the collection are articles.

PlainList uses the values in `c` and `cid` (passed in from the SectionFront element) to load and scatter the collection:

```
<ASSET.LOAD TYPE="Variables.c" OBJECTID="Variables.cid"
NAME="PlainListCollection"/>
<ASSET.SCATTER NAME="PlainListCollection" PREFIX="asset"/>
```

PlainList then sets the variable `ct` to `Full` because a value for this variable was not passed in (`Full` is the name of an article template):

```
<IF COND="IsVariable.ct!=true">
   <THEN>
   <SETVAR NAME="ct" VALUE="Full"/>
   </THEN>
</IF>
```

Next PlainList creates a list of all the child articles in the collection asset, listing them by their rank, and naming the list “theArticles”.

```
<ASSET.CHILDREN NAME="PlainListCollection" LIST="theArticles"
OBJECTTYPE="Article" ORDER="nrank" CODE="-"/>
```

Note that this `ASSET.CHILDREN` tag used the `OBJECTTYPE` parameter. If you use the `OBJECTTYPE` parameter with this tag, the resulting list of children is a join of the

AssetRelationTree and the asset table for the type you specified—in this case, the Article table—and it contains data from both tables.

There is now no need for subsequent ASSET.LOAD tags because the data that the PlainList element is going to use to create the links to these articles is stored in the Article table.

PlainList loops through the list of articles, using the RENDER.GETPAGEURL tag to create a URL for each one. In this case —because the code does not use subsequent ASSET.LOAD tags for each of the children assets— the element includes a RENDER.LOGDEP tag in the loop:

```
<LOOP LIST="theArticles">
      <RENDER.LOGDEP cid="theArticles.id" c="Article"/>
      <RENDER.GETPAGEURL PAGENAME="BurlingtonFinancial/Article/
      Variables.ct"
            cid="theArticles.id"
            c="Article"
            p="Variables.p"
            OUTSTR="referURL"/>
```

PlainList passes a cid and pagename and the asset type with ctype for each article in the collection to the RENDER.GETPAGEURL tag. Because the variable ct was set to Full, the page name being passed to the tag is actually BurlingtonFinancial/Article/Full.

The RENDER.GETPAGEURL tag returns a referURL variable for each article in the collection, as specified by the OUTSTR parameter, and then PlainList uses the value in the referURL variable to create an HTML <A HREF> link for each article.

Because the ASSET.CHILDREN tag that obtained this collection created a join between AssetRelationTree and the Article table, PlainList can use the article's subheadline field to create the link:

```
            <A class="wirelink" HREF="Variables.referURL"
                REPLACEALL="Variables.referURL">
               <csvar NAME="Variables.theArticles:subheadline"/>
            </A>
   </LOOP>
```

Note the use of the REPLACEALL tag as an attribute for this HTML tag. You must use this tag as an HTML attribute when you want to use XML variables in an HTML tags. (For more information about REPLACEALL, see the *Content Server Tag Reference*.)

# Example 3: Using the ct Variable

The ct variable represents the concept of a "child template." Child templates are alternate templates. Because assets are assigned a template when they are created, the identity of an asset's template (which is not the same as a default approval template) is part of the information you obtain with an ASSET.LOAD or an ASSET.CHILDREN tag.

However, sometimes you want to use a template other than an asset's default template. In such a case, you supply the name of an alternate template with the ct variable.

For example, when a visitor browses the Burlington Financial site, there are text-only versions of most of the site available to that visitor. The text-only format is not the default format and content providers do not assign text-only formats to their assets. The

Burlington Financial page elements are coded to provide the ID of the alternate, text-only template when it is appropriate to do so.

Open the Content Server interface, and preview both the Burlington Financial Columnists page and the News Page. In the upper right corner of these pages, the Plain Text link is displayed.

Click the **Plain Text** link on the Columnists page. Then click the Plain Text link on the News Page:

Every page on the site uses the same element, the TextOnlyLink element, to determine the URL embedded in the Plain Text link for that page. The TextOnlyLink element returns the correct URL for each page because the Plain Text link on each page passes the TextOnly element the information that it needs:

- The ID of the page making the request
- The alternate, text-only template (that is, the child template) to use for the Plain Text link

These are the elements used in this example:

- ElementCatalog/BurlingtonFinancial/Page/SectionFront
- ElementCatalog/BurlingtonFinancial/Page/SectionFrontText
- ElementCatalog/BurlingtonFinancial/Common/TextOnlyLink
- ElementCatalog/BurlingtonFinancial/Page/ColumnistFront

## First Element: SectionFront

Use Content Server Explorer to open and examine the Section Front element:

ElementCatalog/BurlingtonFinancial/Page/SectionFront.

SectionFront is the template element (root element) of the Template asset assigned to the standard section pages on the site, pages such as News, Money, Stocks, and so on.

Scroll down approximately two-thirds of the element to this CALLELEMENT tag:

```
<CALLELEMENT NAME="BurlingtonFinancial/Common/TextOnlyLink">
<ARGUMENT NAME="ct" VALUE="SectionFrontText"/>
<ARGUMENT NAME="assettype" VALUE="Page"/>
</CALLELEMENT>
```

TextOnlyLink is the element that creates the Plain Text Link. SectionFront passes it the name of the alternate template (ct="SectionFrontText") and the name of the asset type (assettype="Page").

## Second Element: TextOnlyLink

The TextOnlyLink element executes. Use Content Server Explorer to open and examine it:

ElementCatalog/BurlingtonFinancial/Common/TextOnlyLink

First, TextOnlyLink checks to see whether there is a value for ct.

```
<IF COND="IsVariable.ct!=true">
<THEN>
      <SETVAR NAME="ct" VALUE="Variables.asset:templateText"/>
</THEN>
</IF>
```

There is a value for ct because the SectionFront element passed in ct="SectionFrontText".

Next, TextOnlyLink uses a RENDER.GETPAGEURL tag to obtain a URL for the Plain Text link, passing in the page name by concatenating one based on the variables that were passed to TextOnlyLink by SectionFront.

```
<RENDER.GETPAGEURL PAGENAME="BurlingtonFinancial/
Variables.assettype/Variables.ct"
      cid="Variables.asset:id"
      c="Variables.assettype"
      p="Variables.p"
      OUTSTR="referURL"/>
```

TextOnlyLink knows that ct="SectionFrontText" and that assettype="Page". Therefore BurlingtonFinancial/Variables.assettype/Variables.ct means BurlingtonFinancial/Page/SectionFrontText.

Now that TextOnlyLink has a URL (in the referURL variable specified by the OUTSTR parameter), it can create the Plain Text link with an HTML <A HREF> tag:

```
<A class="contentlink" HREF="Variables.referURL"
REPLACEALL="Variables.referURL">
<img src="/futuretense_cs/bf/images/TextOnly.gif" width="22"
      height="14" border="0" HSPACE="3"/>Plain Text</A><BR/>
```

Note the use of the REPLACEALL tag as an attribute for this HTML tag. You must use this tag as an HTML attribute when you want to use XML variables in an HTML tag. (For more information about REPLACEALL, see the *Content Server Tag Reference*.)

And then `TextOnlyLink` clears the `ct` variable.

```
<REMOVEVAR NAME="ct"/>
```

When a visitor clicks the Plain Text link, the article is formatted with the `SectionFrontText` element and then displayed in the browser.

**Note**

If you are interested in the format of the plain text version of a section page, use Content Server Explorer to open and examine SectionFrontText:

```
ElementCatalog/BurlingtonFinancial/Page/
SectionFrontText
```

## ColumnistFront

Use Content Server Explorer to open and examine the `ColumnistFront` element:

`ElementCatalog/BurlingtonFinancial/Page/ColumnistFront`

This element formats the web format page that displays the stories supplied from the Burlington Financial columnists.

To create the Plain Text link in the upper right corner of a section page, `ColumnistFront` calls `TextOnlyLink`:

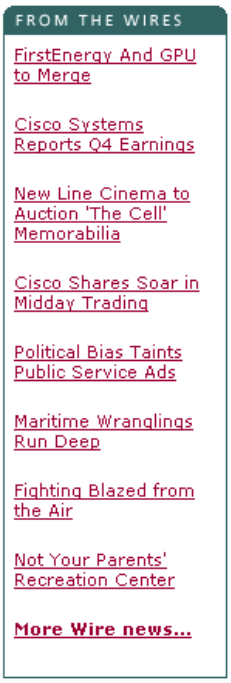
```
<CALLELEMENT NAME="BurlingtonFinancial/Common/TextOnlyLink">
   <ARGUMENT NAME="ct" VALUE="ColumnistFrontText"/>
   <ARGUMENT NAME="assettype" VALUE="Page"/>
</CALLELEMENT>
```

Based on the information passed in from ColumnistFront, this time TextOnlyLink creates a Plain Text link that takes the visitor to `BurlingtonFinancial/Page/ ColumnistFrontText`.

# Example 4: Coding Templates for Query Assets

When you use a query asset to obtain the assets that you want to display, you use the standard CS-Direct element name `ExecuteQuery` to run it.

Burlington Financial uses several query assets. This example describes a query asset named Home Wire Feed which is used to list wire feed stories on the Home page:




These are the elements used in this example:

- ElementCatalog/BurlingtonFinancial/Page/Home
- ElementCatalog/BurlingtonFinancial/Query/WireFeedBox
- ElementCatalog/OpenMarket/Xcelerate/AssetType/Query/ExecuteQuery

## First Element: Home

Use Content Server Explorer to open and examine the template element for the Home page:

`ElementCatalog/BurlingtonFinancial/Page/Home`

First, Home loads the home page asset, names it HomePage, and then scatters the information in its fields:

```
<ASSET.LOAD TYPE="Variables.c" OBJECTID="Variables.cid"
NAME="HomePage"/>
<ASSET.SCATTER NAME="HomePage" PREFIX="asset"/>
```

The values for `c` and `cid` are passed in from the Burlington Financial URL.

Scroll down past several `CALLELEMENT` and `RENDER.SATELLITEPAGE` tags to the following `ASSET.CHILDREN` tag:

```
<ASSET.CHILDREN NAME="HomePage" LIST="WireFeedStories"
```

```
CODE="WireFeed"/>
```

Notice that in this line of code, the `OBJECTTYPE` parameter is not used. `CODE="WireFeed"` is enough information for Content Server Direct to locate and retrieve the query assigned to the Home page asset through the WireFeed association and there is no need to create a join between the `AssetRelationTree` and the `Query` table because all that Home needs is the ID of the query. The WireFeed query is retrieved and stored as "`WireFeedStories`".

Next, Home calls the page entry of the WireFeedBox template, passing it the `cid` of the query stored as "`WireFeedStories`":

```
<RENDER.SATELLITEPAGE PAGENAME="BurlingtonFinancial/Query/
WireFeedBox"
ARGS_cid="WireFeedStories.oid"
ARGS_p="Variables.asset:id"/>
```

Home passes on several pieces of information: the identity of the query with the `cid="WireFeedStories.oid"` statement and the identity of the originating page asset, Home, with the `p="Variables.asset:id"` statement.

## Second Element: WireFeedBox

The WireFeedBox page entry invokes its root element, the template element for the WireFeedBox template. Use Content Server Explorer to open and examine it:

`ElementCatalog/BurlingtonFinancial/Query/WireFeedBox`

This element invokes the `ExecuteQuery` element to run the query and then displays a list of links to the article assets returned by the query.

First, `WireFeedBox` loads the query asset passed in from `Home`, names it `"WireFeedBoxQuery"`, and then retrieves the values from all of its fields with an `ASSET.SCATTER` statement:

```
<ASSET.LOAD TYPE="Variables.c" OBJECTID="Variables.cid"
NAME="WireFeedBoxQuery"/>
<ASSET.SCATTER NAME="WireFeedBox" PREFIX="asset"/>
```

`Variables.cid` is the `WireFeedStories.oid` passed in from the Home element.

Then `WireFeedBox` calls the `ExecuteQuery` element:

```
<CALLELEMENT NAME="OpenMarket/Xcelerate/AssetType/Query/
ExecuteQuery">
<ARGUMENT NAME="list" VALUE="ArticlesFromWireQuery"/>
<ARGUMENT NAME="assetname" VALUE="WireFeedBoxQuery"/>
<ARGUMENT NAME="ResultLimit" VALUE="8"/>
</CALLELEMENT>
```

WireFeedBox passed in the query asset, the name of the list to create to hold the results of the query, and a limit of 8 so that no matter how many assets the query returns to `ExecuteQuery`, `ExecuteQuery` returns only 8 of them to WireFeedBox.

## Third Element: ExecuteQuery

The `ExecuteQuery` element runs the query asset.

The query assets that can be assigned to a page asset as that page's "Wire Feed" query are coded to return field data rather than the IDs of assets only. Therefore, `ExecuteQuery` returns up to eight article assets and the data from several of their fields to `WireFeedBox`.

Use Content Server Explorer to open and examine `ElementCatalog/OpenMarket/Xcelerate/AssetType/Query/ExecuteQuery` if you are interested in this element. Notice that the first line of code in the element is `RENDER.UNKNOWNDEPS` because there is no way of knowing which assets will be returned so there is no way to log dependencies for them.

When `ExecuteQuery` is finished, `WireFeedBox` resumes.

## Back to WireFeedBox

`WireFeedBox` resumes, looping through the list of articles returned by `ExecuteQuery`, and obtaining a URL for each one by using a `RENDER.GETPAGEURL` tag.

Because there is no way of knowing which article assets will be returned by `ExecuteQuery`, there is a `RENDER.FILTER` tag included in the loop to filter out unapproved assets when the publishing method is Export to Disk:

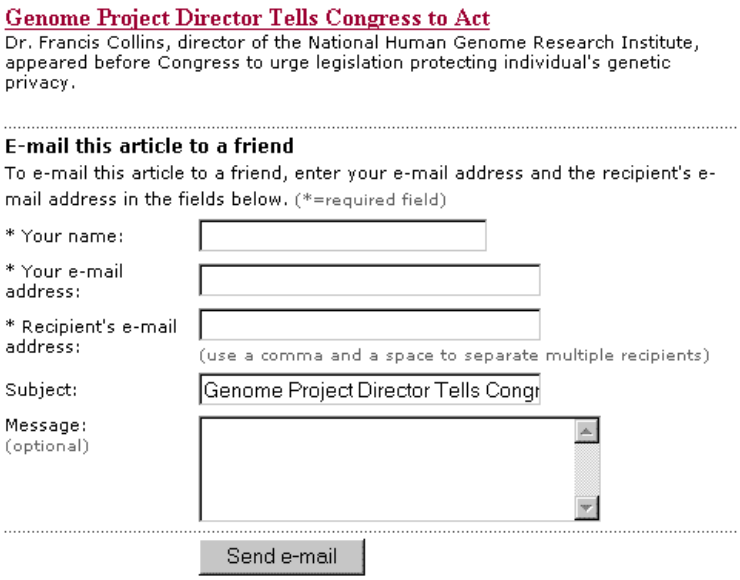
```
<RENDER.FILTER LIST="ArticlesFromWireQuery"
LISTVARNAME="ArticlesFromWireQuery" LISTIDCOL="id"/>
   <if COND="ArticlesFromWireQuery.#numRows!=0">
   <then>
   <LOOP LIST="ArticlesFromWireQuery">
   <RENDER.GETPAGEURL PAGENAME="BurlingtonFinancial/Article/
   WireStory"
   cid="ArticlesFromWireQuery.id"
   c="Article"
   p="Variables.p"
   OUTSTR="referURL"/>
   <A class="wirelink" HREF="Variables.referURL"
   REPLACEALL="Variables.referURL"><csvar
   NAME="ArticlesFromWireQuery.subheadline"/></A><P/>
         </LOOP>
   </then>
   </if>
```

The `RENDER.GETPAGEURL` tag returns a URL for each article in the list in a variable named `referURL`. `WireFeedBox` uses the value from the `referURL` variable to create links to the articles, using the content from their subheadline fields (which is one of the fields that the Wire Feed query returned) as the hyperlinked text.

Note the use of the `REPLACEALL` tag as an attribute for this HTML tag. You must use this tag as an HTML attribute when you want to use XML variables in an HTML tag. (For more information about `REPLACEALL`, see the *Content Server Tag Reference*.)

# Example 5: Displaying an Article Asset Without a Template

Burlington Financial provides an “email this article to a friend” function. Here is the email form for an article:




Obviously the Burlington Financial developers do not want the Burlington Financial content providers to assign the email form to an article as the article’s Display Style (template). Therefore, there is no Template asset that points to the email element that creates the article email form.

These are the elements used in this example:

- ElementCatalog/BurlingtonFinancial/Article/Full
- ElementCatalog/BurlingtonFinancial/Article/AltVersionBlock
- ElementCatalog/BurlingtonFinancial/Util/EmailFront

## First Element: Full

Use Content Server Explorer to open and examine the template element for the `Full` template:

`ElementCatalog/BurlingtonFinancial/Article/Full`

This element provides the formatting code for articles when they are displayed in full. It displays the following items:

- A site banner
- The left navigation column
- A collection of related stories
- The text of the article
- A photo for the article

- A link that prints the story
- A link that emails the story

Scroll down past several RENDER.SATELLITEPAGE and CALLELEMENT tags to the following tag:

```
<CALLELEMENT NAME="BurlingtonFinancial/Article/
AltVersionBlock"/>
```

## Second Element: AltVersionBlock

Use Content Server Explorer to open and examine this element:

ElementCatalog/BurlingtonFinancial/Article/AltVersionBlock

AltVersionBlock is a short element with two RENDER.GETPAGEURL tags. The first RENDER.GETPAGEURL tag obtains the URL for the print version of an article. The second RENDER.GETPAGEURL tag obtains the URL for the email version of the story.

Because the Burlington Financial developers want a dynamic URL for the email version of the story even if the site is a static site, the second RENDER.GETPAGEURL tag uses the DYNAMIC parameter.

Scroll down to the second RENDER.GETPAGEURL tag:

```
<RENDER.GETPAGEURL PAGENAME="BurlingtonFinancial/Util/
EmailFront"
cid="Variables.asset:id"
c="Article"
DYNAMIC="true"
OUTSTR="referURL"/>
```

AltVersionBlock passes in the pagename for the EmailFront page entry, and a value for c, and cid, and sets the DYNAMIC parameter to "true". The tag creates a dynamic URL for the article (even if the publishing method is Export to Disk) and returns it in a variable named referURL, as specified by the OUTSTR parameter.

## Third Element: EmailFront

EmailFront is the pagename that AltVersionBlock passes to the RENDER.GETPAGEURL element. Because there is no corresponding template for EmailFront, CS-Direct did not create a page entry in the SiteCatalog for EmailFront by default. The Burlington Financial developers created the SiteCatalog entry for this element manually through Content Server Explorer.

Use Content Server Explorer to open and examine its root element:

ElementCatalog/BurlingtonFinancial/Util/EmailFront

This element creates a form that displays the first paragraph of the article that the visitor has chosen to email.

First, EmailFront loads the article asset:

```
<ASSET.LOAD TYPE="Article" OBJECTID="Variables.cid"
NAME="EmailFront"/>
<ASSET.SCATTER NAME="EmailFront" PREFIX="asset"/>
```

Then it formats several parts of the page before creating the email form. Scroll down to the HTML FORM tag:

```
<FORM NAME="mailform" onSubmit="return checkEmail();"
METHOD="POST" ACTION=...
```

EmailFront then calls the `LeadSummary` page entry to display a summary of the article in the form:

```
<RENDER.SATELLITEPAGE
ARGS_pagename="BurlingtonFinancial/Article/LeadSummary"
ARGS_cid="Variables.cid"
ARGS_ct="Full"
ARGS_p="Variables.p"/>
```

For information about the `LeadSummary` element, see "Example 1: Basic Modular Design" on page 620 or use Content Server Explorer to open and examine it.

# Example 6: Displaying Site Plan Information

Because the developers of the Burlington Financial sample site used the Site Plan tab in the Content Server interface to order the basic structure of the Burlington Financial site, they are able to extract information from the `SitePlanTree` table to create navigational features for the site.

For example, the navigation bar at the top of the Burlington Financial home page is created by extracting information about the site's structure from the `SitePlanTree` table.

To extract information from the `SitePlanTree` table, you use the CS-Direct `SITEPLAN` tag family.

These are the elements used in this example:

- ElementCatalog/BurlingtonFinancial/Article/Home
- ElementCatalog/Pagelet/Common/SiteBanner
- ElementCatalog/BurlingtonFinancial/Site/TopSiteBar

## First Element: Home

Use Content Server Explorer to open and examine the template element for the Home template:

`ElementCatalog/BurlingtonFinancial/Page/Home`

Scroll down to the first `RENDER.SATELLITEPAGE` tag:

```
<RENDER.SATELLITEPAGE PAGENAME="BurlingtonFinancial/Pagelet/
Common/SiteBanner"/>
```

## Second Element: SiteBanner

The `SiteBanner` pagelet invokes its root element. Use Content Server Explorer to open and examine it:

`ElementCatalog/BurlingtonFinancial/Common/SiteBanner`

SiteBanner gathers the images for the banner (the Burlington Financial logo and an advertising image) and then calls an element that creates the navigational links to the main sections of the site.

Scroll down to this CALLELEMENT tag:

```
<CALLELEMENT NAME="BurlingtonFinancial/Site/TopSiteBar"/>
```

## Third Element: TopSiteBar

TopSiteBar executes, creating the navigational links to the main sections in the site. Use Content Server Explorer to open and examine TopSiteBar:

ElementCatalog/BurlingtonFinancial/Site/TopSiteBar

### Creating the Link for the Home Page

First, TopSiteBar loads the Home page, names it "target", gets the value from its ID field, and stores that value in the output variable "pageid":

```
<ASSET.LOAD TYPE="Page" NAME="target" FIELD="name" VALUE="Home"
DEPTYPE="exists"/>
<ASSET.GET NAME="target" FIELD="id" OUTPUT="pageid"/>
```

Note that the ASSET.LOAD tag changes the dependency type from its default of exact to exists with the DEPTYPE parameter. For a link like this one, a link in a navigational bar, it makes more sense for the dependency to be an exists dependency.

Then TopSiteBar uses the variable pageid to obtain a URL for the Home page from a RENDER.GETPAGEURL tag:

```
<RENDER.GETPAGEURL PAGENAME="BurlingtonFinancial/Page/Home"
            cid="Variables.pageid"
            c="Page"
            OUTSTR="referURL"/
```

Next TopSiteBar extracts the page asset's name from its **Name** field and uses it as the text for the hyperlink:

```
<ASSET.GET NAME="target" FIELD="name" OUTPUT="thepagename"/>
<A class="sectionlinks" HREF="Variables.referURL"
REPLACEALL="Variables.referURL"><csvar
NAME="Variables.thepagename"/></A>
```

Note the use of the REPLACEALL tag as an attribute for this HTML tag. You must use this tag as an HTML attribute when you want to use XML variables in an HTML tag. (For more information about REPLACEALL, see "Using Variables in HTML Tags" on page 110.)

### Creating the Links to the Home Page's Child Pages

In the next part of the code, TopSiteBar creates links for the child pages of the Home page. In order to determine the child pages of the Home page, it must first determine the node ID of the Home page.

The node ID of a page asset is different from its object ID:

- You use an object ID to extract information about an asset from asset tables.
- You use a node ID to extract information about a page asset from the SitePlanTree table.

First, TopSiteBar determines the node ID of the Home page:

```
<ASSET.GETSITENODE NAME="target" OUTPUT="PageNodeId"/>
```

Then it uses that information to load the Home page as a siteplan node object:

```
<SITEPLAN.LOAD NAME="ParentNode" NODEID="Variables.PageNodeId"/>
```

With the Home page node identified and loaded, TopSiteBar can then obtain the Home page's child nodes, storing them in a list that it names "`PeerPages`," and ordering them according to their rank:

```
<SITEPLAN.CHILDREN NAME="ParentNode" TYPE="PAGE" LIST="PeerPages"
CODE="Placed" ORDER="nrank"/>
```

And now `TopSiteBar` loops through all the child nodes at the first level, using the `RENDER.GETPAGEURL` tag to create a URL for the link to each page:

```
<IF COND="IsList.PeerPages=true">
<THEN>
   <LOOP LIST="PeerPages"> | 
         <ASSET.LOAD NAME="ThePage" TYPE="Page"
             OBJECTID="PeerPages.oid"/>
            <ASSET.GET NAME="ThePage" FIELD="name"
               OUTPUT="thepagename"/>
            <ASSET.GET NAME="ThePage" FIELD="template"
               OUTPUT="pagetemplate"/>
         <RENDER.GETPAGEURL PAGENAME="BurlingtonFinancial/Page/
         Variables.pagetemplate"
                     cid="PeerPages.oid"
                     c="Page"
                     OUTSTR="referURL"/>
      <A class="sectionlinks" HREF="Variables.referURL"
   REPLACEALL="Variables.referURL">
   <csvar NAME="Variables.thepagename"/>
   </A>
```

Notice how the page name is constructed in this example. The second `ASSET.GET` statement in the preceding piece of code obtains the name of the page's template from its template field. Here it is again:

```
   <ASSET.GET NAME="ThePage" FIELD="template"
   OUTPUT="pagetemplate"/>
```

Then, that information is used in the `PAGENAME` parameter passed to the `RENDER.GETPAGEURL` tag:

```
   PAGENAME="BurlingtonFinancial/Page/Variables.pagetemplate"/>
```

Therefore, if the template for the page asset is SectionFront, this argument statement passes `pagename="BurlingtonFinancial/Page/SectionFront`. And if the template for the page asset is AboutUs, this argument statement passes `pagename="BurlingtonFinancial/Page/AboutUs.`

## Back to SiteBanner

SiteBanner is finished after the call to `TopSiteBar`. The `SiteBanner` element is invoked on each page in the site.

Because `SiteBanner` has a page entry in the `SiteCatalog` table, the results of the navigational bar that `TopSiteBar` creates is cached the first time a visitor requests a page on the Burlington Financial site. This speeds up performance because the site does not have to re-invoke the `TopSiteBar` element for each and every page that the visitor subsequently visits.

# Example 7: Displaying Non-Asset Information

Sometimes you need to render and display information that is not stored as an asset in the Content Server database. For example, the Burlington Financial site displays today's date on each page. The date is not information that can be stored as an asset.

These are the elements used in this example:

- ElementCatalog/BurlingtonFinancial/Article/Home
- ElementCatalog/Common/ShowMainDate

## First Element: Home

Use Content Server Explorer to open and examine the template element for the Home template:

`ElementCatalog/BurlingtonFinancial/Page/Home`

Scroll down to the third `CALLELEMENT` tag, one that invokes the `ShowMainDate` element.

```
<CALLELEMENT NAME="BurlingtonFinancial/Common/ShowMainDate"/>
```

## Second Element: ShowMainDate

`ShowMainDate` executes. Use Content Server Explorer to open and examine it:

`ElementCatalog/BurlingtonFinancial/Common/ShowMainDate`

The main line of code is this one:

```
<span class="dateline"><csvar NAME="CS.Day CS.Mon CS.DDate,
CS.Year"/></span>
```

It calculates the date and then returns that value to the `Home` element, which displays it at the top of the page, under the navigation bar and over the main list of stories.

This element performs a simple calculation and then outputs the value into the HTML code that is rendered in the browser window. There are no content assets that it formats or Template assets that use it as a root element. It also has no `SiteCatalog` entry because its result—the date—should be calculated each time the Home page is rendered.

Chapter 29

# Configuring Sites for Multilingual Support

This chapter explains how to configure multilingual support for a site.

This chapter contains the following sections:


- Overview
- Working with Locale Filtering
- Planning Multilingual Support for a Site
- Configuring Multilingual Support for a Site

# Overview

When you configure a site for multilingual support, users in that site gain the ability to assign **locale** (language version) designations to assets, and to create translations of assets. You also have the option to create site-specific delivery rules for multilingual content that determine which language versions of assets will be shown on the online site, and what to do if a visitor's request is for a language version in which the content does not yet exist.

This chapter describes important topics you need to consider when planning and implementing the design of your site with respect to multilingual support. The chapter then goes on to describe the procedures necessary to configure your site to support multilingual assets and related features.

## Dimensions

Locale designations in Content Server are implemented through the concept of **dimensions**. A dimension is an identifier that differentiates assets that are otherwise semantically identical. A locale (such as `en_US` for US English) is thus a type of dimension that differentiates two translations of the same content.

Dimensions are represented by assets of type "Dimension." This asset type must be enabled by developers on a per-site basis so that users can create "Dimension" assets (of subtype "Locale"), and so that content providers can assign locale designations to assets they wish to translate.

**Note**

Users cannot create translations of assets that have no locale designation assigned.

Each "Dimension" asset represents a locale in the site – for example, an `en_US` "Dimension" asset represents US English, and an `fr_CA` "Dimension" asset represents Canadian French. When a content asset is assigned a locale, the assignment is recorded in the *`assetType`*`_Dim` table for the corresponding asset type.

Publishing content in a given locale requires enabling the locale on the online site, that is, publishing the "Dimension" asset representing the locale to the delivery system, and including the locale in the site's dimension set (see "Dimension Sets" below.)

## Dimension Sets

When you have created your locales, we recommend that you create at least one dimension set. A dimension set is a grouping of dimension assets (locales), which affects the delivery of content to the site visitor in the following ways:

- A dimension set defines which locales are designated as enabled for the online site. In other words, a dimension set determines the languages in which content will be shown to the visitors. For example, one dimension set would contain European languages, another Asian languages, and so on.
- A dimension set defines to filter content based on locale – for example, how to handle content that does not exist in the visitor's preferred language at the time the visitor requests it. If you do not publish a dimension set to the delivery system, locale filtering will not function. (For information on locale filtering, see "Working with Locale Filtering," on page 647.)

**Note**

Dimension sets do not affect the operation of Content Server user interfaces.

For locale filtering to function on the online site, you must approve and publish to the delivery system both the “DimensionSet” assets and the “Dimension” assets referenced by each dimension set. (Because referenced “Dimension” assets are dependents of the “DimensionSet” assets referencing them, they must be approved with their respective “DimensionSet” assets.)

## Cross-Site Multilingual Support

If you are setting up multilingual support in more than one site, you can choose to implement one of the following scenarios:

- **Create the locales, but no dimension sets**

  This option allows content providers to manage content in multiple languages, but does not enable render-time locale filtering. Use this option only if translations in all required languages will exist for each asset from the very beginning.

- **Create the locales and a dimension set, and share them across your sites**

  This option provides the simplest way of enabling multilingual support on multiple sites. Sites set up in this way share the properties stored in the dimension set (locales enabled for display on the online site, the locale filtering method, and, if applicable, the fallback hierarchy – the path the Hierarchical filter traverses when looking up asset translations at render time; see “The Hierarchical Filter,” on page 649 for details). If you are creating a “bare-bones” site that you will replicate into multiple target sites, it is best to share your locales to the target sites.

- **Create separate locales and dimension sets for each site**

  This option affords the most flexibility, at the cost of increased configuration complexity. Sites set up this way benefit from the fact that properties such as locale filter type or fallback hierarchy can be tailored to each site.

  **Note**

  While creating duplicate “Dimension” assets to represent the same language in multiple sites is possible, it is not recommended, as it introduces unnecessary complexity.

- **A mixture of the latter two options**

  This option provides the right balance between flexibility and configuration complexity. As a possible best practice, you would create a pool of unique locales, share the locales required by each site from that pool, and share or create dimension sets for each site as needed.

## Master Assets, Translations, and Multilingual Sets

When an asset is assigned a locale for the first time, it gains **master**, or **dimension parent**, status. Master status allows the formation of a multilingual set – a group of assets whose content is semantically identical, but exists in different languages. (Note that this is not the same as a dimension set; a dimension set only affects the online site and the way assets are rendered.)

**Note**

The terms “master asset” and “dimension parent” are equivalent. “Master asset” is displayed in Content Server’s user interfaces; “dimension parent” is used in the tag reference, database table names (such as `assetType_DimP`), and element code.

For example, when you designate an “Article” asset as US English (`en_US`), and create translations of it in whichever locales are enabled in the site (such as French (`fr_CA`) and German (`de_DE`)), the translations point to their dimension parent – the US English asset – to indicate they are semantically equivalent to the master and one another.

When you create a translation of an asset with master status, Content Server copies the asset and assigns the locale of your choice to the copy. You then enter the translated content and save the translation as a new asset.

At this point, the source asset and its translation are linked into a multilingual set, and the translation adopts the source asset as its master, or dimension parent. Any member of the set that is not the master can be given master status; however, only one set member at a time can be the master.

The linking is accomplished through the `assetType_DimP` table for the asset type. The table stores the following information:

- ID of the master (dimension parent) asset
- ID of the translation asset
- ID of the locale dimension asset assigned to that translation

Even though Content Server interfaces allow you to initiate the creation of a translation from either the master asset or any of its existing translations, all translations in the multilingual set always point to the dimension parent (master) asset.

**Note**

If a locale-aware asset is being revision-tracked, changes to asset locale data (such as locale designation or master status) do not generate a new version of the asset.

## Translations and Asset Relationships

The way asset relationships are handled when an asset is translated is summarized in the following table:

| Relationship Type | Behavior |
|---|---|
| Named and Unnamed Associations | When an asset containing named or unnamed associations is translated, all assets associated with the source asset are automatically associated with the translation. You then have the choice to translate the associated assets and associate the translated versions with the translated parent asset. |
| Collections | When you create a translation of a “Collection” asset, the new “Collection” asset retains the member assets of the source asset. You then have the choice to translate the member assets and place the translated versions in the new “Collection” asset, replacing the member assets carried over from the old collection. |
| Static Lists Recommendations | When you create a new language version of a Static Lists recommendation, the new “Recommendation” asset retains the member assets of the source asset. You then have the choice to translate the member assets and place the translated versions in the new “Recommendation” asset, replacing the member assets carried over from the old collection. |
| Dynamic Lists Recommendations | Since Dynamic Lists recommendations are populated by element code, they are not affected. |
| Related Items Recommendations | When an asset containing Related Items associations is translated, all assets associated with the source asset are automatically associated with the translation. You then have the choice to translate the associated assets and associate the translated versions with the translated parent asset. |
| Asset-Type Attributes | When an asset containing associations through asset-type attributes is translated, all assets associated with the source asset are automatically associated with the translation. You then have the choice to translate the associated assets and associate the translated versions with the translated parent asset. |
| Embedded Links | Embedded links are not affected. When an asset containing embedded links is translated, you must manually update the links to point to the corresponding translations of the linked content (if such translations exist). |

For information on handling asset relationships at render time, see “Working with Locale Filtering,” on page 647.

## Approval Dependencies

An approval dependency exists between two assets when editing one of the assets causes the other's approval status to change. The following table summarizes the approval dependencies affecting localized assets:

| Dependency | Effect on Asset Approval |
|---|---|
| An "Exists" dependency exists between a localized asset and the "Dimension" asset representing the assigned locale. | To approve a localized asset for publishing, the corresponding "Dimension" asset must also be approved. |
| In a multilingual set, an "Exists" dependency exists between the master asset and each translation linked to it. | When you create the first translation of an asset, you must approve both the asset and its translation.<br>To approve a translation, you must also approve the corresponding master asset, unless the master asset has already been approved.<br>You must reapprove all members of the set if:<br>• You add a new translation to, or delete an existing translation from the set.<br>• You edit the set's master asset.<br>• You designate another member of the set as the master. |
| An "Exists" dependency exists between a "DimensionSet" asset and the "Dimension" assets representing the locales enabled in that dimension set. | To approve a dimension set, the corresponding "Dimension" assets must also be approved. |

# Working with Locale Filtering

Ideally, a multilingual site would be complete in terms of its content before it is launched — that is, all assets and their relatives would exist in all of the required languages. However, in many situations, this is not so. In such cases, you would use a locale filter to decide at render time which language version of an asset to show on the site depending on the circumstances and the language preference specified by the visitor. For example, you could let the business logic decide what to do if a requested asset does not exist in the requested language.

By using locale filtering, you can also spread editorial work over time by allowing content providers to create the required translations after the original content is published to the online site. Locale filtering allows the site to automatically "pick up" the missing translations as soon as they are published to the delivery system.

Keep in mind that locale filtering introduces additional load on the delivery system. The amount of additional load depends on the complexity of the filtering logic.

## Handling Asset Relationships Through Locale Filtering

The way you choose to implement locale filtering will have an influence on how asset relationships on your site are structured, and vice versa, depending on the way you want the online site to behave.

You can choose to implement one of the following options:

- **Maintain different asset relationship trees for each locale**

  When rendering assets, this model renders whatever assets are associated with the requested asset.

  For example, if an asset exists in English and French, and each version has a unique set of associated assets, each version is rendered with its respective associated assets. Filtering is only used to look up and deliver a version of the requested asset matching the language preference specified by the visitor; the associated assets are expected to already have been translated into all required languages.

  This model allows for completely independent content for each language. It is used in the First Site II sample site.

- **Use the same asset relationship tree for all locales**

  When rendering assets, this model substitutes the associated assets of the requested asset with the assets associated with a specific language version of the requested asset, regardless of the language preference specified by the visitor.

  For example, if an asset exists in English and French, each version has a unique set of associated assets, and the visitor specified French as their language preference, filtering will look up and deliver the French version of the requested asset, but it will substitute the associated assets of the English version in place of those of the French version (assuming the language version from which filtering is to derive associations is English).

  This model ensures consistent content across all languages.

- **A mixture of the two models**

  Allows for the greatest amount of flexibility and customization for your site. The optimal proportion between the two models will depend on the intended behavior of your site.

## Included Locale Filters

Content Server ships with the following locale filters:

- The Simple Filter
- The SimpleLookup Filter
- The Hierarchical Filter (also known as the Fallback filter)

You also have the option to implement custom locale filters, if desired. See "Custom Locale Filters," on page 650 for more information on custom filters.

Note that depending on the type of filter you choose to implement, the assets being filtered must satisfy one, or both of the following conditions:

- Assets must have locale designations assigned. Assets without locale designations will be ignored by locale filters.
- Assets that are translations of one another must be linked into multilingual sets (that is, designated as translations of one another through a master asset). Otherwise, the filters will not be able to perform the necessary translation lookups.

### The Simple Filter

The Simple filter is a possible choice for a site that should only be rendered in one language, but whose content exists in multiple languages. The filter checks the following:

- Whether the requested asset is in the language specified by the visitor
- Whether the locale of the asset is listed in the site's dimension set

If both conditions are met, the filter passes the asset to the template for rendering; otherwise, nothing is rendered.

The Simple filter has the least impact on delivery system performance, but increases the amount of editorial work that needs to be done, as assets must exist in the required language versions or they will not be displayed on the online site.

### The SimpleLookup Filter

The SimpleLookup filter is ideal for a site that should only be rendered in one language, but whose content may exist in multiple languages, and for which there is no guarantee that all of the necessary translations exist at render time. The filter checks the following:

- Whether the requested asset is in the language specified by the visitor
- Whether the locale of the asset is listed in the site's dimension set

If the requested asset is not in the visitor's preferred language, the filter looks up a suitable replacement by checking the asset's translations. If the filter finds a matching translation, it passes it to the template; otherwise, nothing is rendered. (The filter will also return nothing if the locale of the translation is not included in the site's dimension set.)

This filter offers a reasonable balance between performance and functionality. While the lookup queries slightly increase the load on the delivery system, the amount of editorial work done to create assets can be reduced, as the required translations can be created after the original content is published to the online site. The lookup mechanism will "pick up" the missing translations as soon as they are published to the delivery system.

The FirstSite II sample site uses this filter as the default locale filter.

## The Hierarchical Filter

The Hierarchical filter checks whether the locale of the requested asset matches the locale requested by the visitor. If the locales do not match, the filter checks the asset's translations to see if a suitable replacement exists. If the filter finds a matching translation, it passes it to the template; otherwise, it substitutes translations of the requested asset according to the fallback hierarchy you set up when you configure the site's dimension set. The fallback hierarchy determines which language versions the filter should substitute for the requested asset, and in what order.

For example, consider the following hierarchy:

- **en_US** (US English)
  - **de_DE** (German)
    - **de_CH** (Swiss German)
    - **de_AT** (Austrian German)
  - **fr_FR** (French)
    - **fr_BE** (Belgian French)
    - **fr_CA** (Canadian French)
  - **en_UK** (British English)

In our example, when the visitor requests an asset in Swiss German (de_CH), the filter looks up the asset's translations and if it finds a Swiss German version of the asset, it passes that version to the template. If the filter cannot find a Swiss German version, it falls back to the "next best" locale in the hierarchy path, German (de_DE). If, in turn, no German translation exists, the filter follows the path specified in the hierarchy until it reaches the top of the tree. If no match is found in the process, nothing is rendered.

Note that the above example describes a situation in which the visitor specifies a single preferred language. If the user specifies multiple preferred languages (in most cases, in the form of an ordered list), the filter attempts to find a match in the fallback hierarchy for the visitor's most preferred language. If no match is found, the filter checks the next language on the visitor's list, until a match in the fallback hierarchy is found. When that happens, the filter attempts to substitute translations of the requested asset by tracing a path from the matching locale to the top of the fallback tree, as described earlier.

For example, if the user's preferred languages are Japanese, French, and English (in that order), the filter attempts to locate Japanese in its fallback hierarchy. Since Japanese is not in the hierarchy, the filter then attempts to locate French. French is in the hierarchy, so the filter traces a path from French to the root node of the tree, and attempts substitution according to that path, as illustrated by the example earlier in this section.

While powerful and convenient, the hierarchical filter has the following drawbacks:

- The additional database queries run by the filter tax the performance of the delivery system. To minimize the performance hit, editorial work should be done to ensure that content exist in as many of the required languages as possible, so that the filter's activity is minimized. (You may also choose to use a different filter.)
- Control over which assets to display on the online site is put exclusively in the hands of the site developer or administrator. This is because the filter follows the fallback tree configured in the dimension set, rather than the preference order specified by the site visitor (assuming the site is set up to accept multiple language preferences from each visitor).

## Custom Locale Filters

Depending on the design of your site, you may decide to create custom filters. For example, your site design might call for a hierarchical (fallback) filter that favors the locale priority specified by the visitor, rather than the one defined in the dimension set. In such cases, a field in the "Edit" form for the "DimensionSet" asset allows you to specify a custom filter class.

## Compositional Dependencies

If you decide to incorporate locale filtering on your site, you must account for the additional compositional dependencies that are introduced as a result. Compositional dependencies determine how pages are cached on your delivery system.

### Asset Lookup Chain

When using locale filtering to look up a translation of an asset, the following factors determine how pages are cached, based on which assets are loaded during the lookup process:

- The filtering logic employed
- The page and asset from which the lookup request originates
- The language preference specified by the visitor

A cached page containing the requested asset is dependent on all assets loaded during the lookup process. Thus, if an asset that is loaded during the lookup process is modified, the affected page is flushed from the cache.

For example, consider the "SimpleLookup" filter and the following multilingual set:

- **en_US** (master)
  - **fr_FR** (translation)
  - **de_DE** (translation)

If a visitor requests a page containing the French version, but the visitor's language preference is German, the lookup chain is as follows:

**fr_FR → en_US → de_DE**

In this example, all three assets are loaded, because the filter must first load the master asset linked to the French version, and then use the master asset to look up the German version. Thus, if any of these three assets is modified, the affected page is flushed from the cache.

If, on the other hand, the user requested the US English version, which is the master asset of the set, then the lookup chain would be shorter:

**fr_FR → en_US**

In such case, the French and US English versions are loaded, but the German version is not. Thus, modifying the German version would not cause the corresponding page to be flushed from the cache, but modifying the French or US English versions would.

For a detailed explanation of the lookup mechanisms employed by locale filters included with Content Server, see "Included Locale Filters," on page 648.

The next section, "Caching Rules," on page 651, explains the caching rules applicable to multilingual assets.

## Caching Rules

Once the translation lookup occurs and the affected page is cached, the page is flushed from the cache whenever one of the following occurs:

- A new language is added to the multilingual set
- A translation that was part of the lookup chain when the page was rendered is edited
- A translation that is a member of the multilingual set is deleted
- The set's master asset is edited
- Another member of the set is designated as the master

# Adding Filtering Support to Your Site

To add support for locale filtering to your site, you must modify the templates and element code used on your site.

When the template code fetches an asset via the asset's `c/cid` values, the locale filter executes its business logic on the incoming `c/cid` values and returns the resulting `c/cid` values (or nothing) to the template for rendering.

The structure of your site will influence how you implement locale filtering in your templates, and vice versa. It will also determine the behavior of your site in different scenarios.

For example, imagine five articles, each existing in two languages, `en` (English), and `fr` (French). The articles would be `a1en`, `a1fr`, `a2en`, `a2fr`, and so on. We can decide to put these articles into a collection and implement locale filtering in one of the following ways:

- Create an English collection, `c1en`, and assign all of the English articles to it. This way, before we render the collection, we would simply filter the `c/cid` of the `c1en` asset, then render its children without filtering their `c/cid` values, because we trust the `c1en` collection to be in a single language.
- Create a multilingual collection (without assigning a locale to it) and add the articles in whatever languages are desired. Then, when rendering each article, filter the article `c/cid` values so that the article is rendered in the locale specified by the visitor.

The rest of this section provides code examples based on the FirstSite II sample site. We recommend that you examine the FirstSite II code to get an idea of how it multilingual support.

## Adding Filtering to Templates

Usually, you would place your filter code into a utility element and call the element at the top of the template to process the `c/cid` values.

The following example shows how the `FSIILayout` template calls the filter code stored in the `FSIICommon/Multilingual/Filter` element asset via the `render:lookup` tag:

```
<%-- Execute the Dimension filter to look up the translated asset
that corresponds to the locale that the visitor requested. --%>
<render:lookup site='<%=ics.GetVar("site")%>' varname="Filter"
   key="Filter" match=":x" tid='<%=ics.GetVar("tid")%>' />
<render:callelement elementname='<%=ics.GetVar("Filter")%>'
   scoped="global"/>
```

## Obtaining and Maintaining a Visitor's Locale Preference

For filtering to work, you have to allow the visitor to specify a preferred language (locale). This preference must then be propagated throughout the entire site (that is, passed to all the templates).

The example below shows how this is accomplished in the `FSIIWrapper` element. The last section of this example shows how the locale variable is set by taking the value from the session variable (earlier in the example, we ensure that the session variable exists).

```
<%-- The session variable locale refers to the id of the dimension
with the subtype of Locale that specifies which language the site
is to be rendered in. Users can select the locale of their choice
from a menu on every page of the site, and once selected, it is
stored in session. A default locale is mapped to this CSElement and
is set if it has not already been set. --%>
<ics:if condition='<%=ics.GetSSVar("preferred_locale") == null%>'>
<ics:then>
<render:lookup site='<%=ics.GetVar("site")%>'
   varname="default:locale:name" key='DefaultLocale'
   ttype="CSElement" tid='<%=ics.GetVar("eid")%>' match=":x"/>
<asset:load name="defaultLocale" type="Dimension" field="name"
   value='<%=ics.GetVar("default:locale:name")%>'/>
<asset:get name="defaultLocale" field="id"
   output="default:locale:id"/>
<ics:setssvar name="preferred_locale"
   value='<%=ics.GetVar("default:locale:id")%>'/>
</ics:then>
</ics:if>
<%-- Call the wrapped child page.  There is no need to look up the
template or to enable any special PageBuilder functionality, so we
can use the render:satellitepage tag in this situation. --%>
<render:satellitepage pagename='<%=ics.GetVar("childpagename")%>'
   packedargs='<%=ics.GetVar("packedargs")%>'>
   <render:argument name='c' value='<%=ics.GetVar("c")%>'/>
   <render:argument name='cid' value='<%=ics.GetVar("cid")%>'/>
   <render:argument name='p' value='<%=ics.GetVar("p")%>' />
   <render:argument name="locale"
      value='<%=ics.GetSSVar("preferred_locale")%>'/>
</render:satellitepage>
```

## Filtering Search Results

If your online site contains search functionality, you may choose to filter the search results returned to the visitor, based on the visitor's language preference.

The following example shows how the `Page/SearchDetailView` template filters an `IList` of search results so that the query can go against all languages but only return the desired results:

```
<%-- look up the dimension set and filter the ProductList results -
-%>
<asset:load name="GlobalDimSet" type="DimensionSet" field="name"
   value='<%=ics.GetVar("GlobalDimSet")%>' />
```

```
<dimensionset:filter name="GlobalDimSet" tofilter="ProductList"
   list="ProductList">
   <dimensionset:asset assettype="Dimension"
      assetid='<%=ics.GetVar("locale")%>'/>
</dimensionset:filter>
```

For more information, see the *Content Server Tag Reference*. Additionally, examine the code in the FirstSite II sample site to see how it implements multilingual support.

# Planning Multilingual Support for a Site

Before you start configuring a site for multilingual support, make sure you have satisfied the following prerequisites. It is best to make these decisions in agreement with your site administrators.

1. Determine how many languages to initially implement on your site (or sites), based on your organization's content management needs. You can either:
   - Build a site (or sites) to support a single language initially, and add support for additional languages as the need arises.
   - Plan ahead for all the languages you expect to incorporate across all of your sites and create the appropriate locales in advance.
2. Determine whether you will share existing locales and dimension sets to the new site (or sites), or create separate ones. For more information, see "Cross-Site Multilingual Support," on page 643.
3. Decide how asset relationships are going to be handled at render time with respect to locales, and choose the locale filtering method(s) appropriate to your decision. The choices you make will have to strike a balance between the desired levels of automation, delivery system performance, and editorial workload, and are thus best made in agreement with your site administrators. For more information, see "Working with Locale Filtering," on page 647. Note the following:
   - Different filtering methods provide different levels of automation at the cost of a performance hit on the delivery system. The more complex the filter, the higher the performance hit. For example, the SimpleLookup filter provides better performance than the Hierarchical filter.
   - Depending on the filtering method you implement, editorial work on site content can be spread over time, as translations can be created after the original content is created and published to the live site. The filtering logic you implement will decide what to do content that does not yet exist in the required language versions.
4. If you are converting a monolingual site to a multilingual site, obtain the element code you will use to assign the default locale to the assets in the site. Sample code based on the FirstSite II sample site is provided in the section, "Sample Element Code for Bulk-Assigning a Default Locale," on page 663.

**Note**

When replicating a site containing multilingual sets, make sure the master assets are available on the target site (by either sharing or copying). Otherwise, the set members will no longer be linked as translations of each other on the target site.

# Configuring Multilingual Support for a Site

This section describes the procedures necessary to configure a site for multilingual support.

The following topics are covered in this section:

- Configuration Quick Reference
- Enabling the "Dimension" and "DimensionSet" Asset Types
- Enabling the "Locale" Subtype of the "Dimension" Asset Type
- Creating a Locale
- Sharing a Locale to Another Site
- Creating and Configuring a Dimension Set
- Sharing a Dimension Set to Another Site
- Configuring a Locale Filter
- Configuring the Fallback Hierarchy of the Hierarchical Filter
- Bulk-Assigning a Default Locale to Assets in a Site

## Configuration Quick Reference

This section provides an overview of the steps necessary to configure multilingual support for a site. Use this list as a quick reference during the configuration process.

**To configure multilingual support for a site**

1. Make the necessary decisions and preparations as described in "Planning Multilingual Support for a Site," on page 653.
2. Enable the "Dimension" and "DimensionSet" asset types on the site. For instructions, see "Enabling the "Dimension" and "DimensionSet" Asset Types," on page 655.
3. Enable the "Locale" subtype of the "Dimension" asset type on the site. For instructions, see "Enabling the "Locale" Subtype of the "Dimension" Asset Type," on page 656.
4. Create or share the desired locales. For instructions, see the following sections:
   - For creating new locales, see "Creating a Locale," on page 656.
   - For sharing existing locales, see "Sharing a Locale to Another Site," on page 657.

   For help in determining whether to create new locales or share existing ones, see "Cross-Site Multilingual Support," on page 643.
5. Create or share a dimension set. For instructions, see the following sections:
   - For creating a new dimension set, see "Creating and Configuring a Dimension Set," on page 658.
   - For sharing an existing dimension set, see "Sharing a Dimension Set to Another Site," on page 659.

   For help in determining whether to create a new dimension set or share an existing one, see "Cross-Site Multilingual Support," on page 643.

6. (Optional) If you are converting an existing monolingual site to a multilingual site, execute element code that assigns a default locale to each asset in the site. For instructions, see "Bulk-Assigning a Default Locale to Assets in a Site," on page 662. The section includes sample code which you can customize for your site.
7. Modify the templates used in the site to include support for the locale filter you selected when you configured the dimension set. For an overview of the process, see "Adding Filtering Support to Your Site," on page 651.

**Note**

In addition to locale filtering, you will have to implement the following site functionality:

- Allow the visitor to specify their language preference
- Propagate the visitor's language preference throughout the site (by passing it to all templates on the site)
- Maintain the visitor's language preference for the duration of the session.

## Enabling the "Dimension" and "DimensionSet" Asset Types

Before you can create "Dimension" and "DimensionSet" assets in your site, you must enable the corresponding asset types and subtypes. This procedure describes how to enable the "Dimension" and "DimensionSet" asset types on your site. The next procedure describes how to enable the "Locale" subtype of the "Dimension" asset type on your site.

**To enable the "Dimension" and "DimensionSet" asset types on your site**

1. Log in to the Content Server interface and select the site in which you want to enable the asset types.
2. In the tree, select the **Admin** tab.
3. In the **Admin** tab, drill down the following hierarchy:
   a. Expand the **Sites** node.
   b. Under the **Sites** node, expand the node corresponding to the desired site.
   c. Under the desired site node, expand the **Asset Types** node.
   d. Under the **Asset Types** node, double-click the **Enable** node.

   Content Server displays the "Enable Asset Types" form.
4. In the "Enable Asset Types" form, select the check boxes next to the **Dimension** and **DimensionSet** asset types.
5. Click **Enable Asset Types**.

   Content Server displays the "Start Menu Selection" form.
6. In the "Start Menu Selection" form, select all of the check boxes, and click **Enable Asset Types**.

Content Server displays a message confirming the asset types have been enabled for the site.

```
Enable Asset Types

Enabled asset type: Dimension
Enabled asset type: DimensionSet
```

## Enabling the “Locale” Subtype of the “Dimension” Asset Type

Before you can assign locales to assets in your site, you must enable the “Locale” subtype of the “Dimension” asset type on your site.

**To enable the “Locale” subtype of the “Dimension” asset type on your site**

1. Log in to the Content Server interface and select the site in which you want to enable the subtype.
2. In the tree, select the **Admin** tab.
3. In the **Admin** tab, drill down the following hierarchy:
   a. Expand the **Asset Types** node.
   b. Under the **Asset Types** node, expand the **Dimension** node.
   c. Under the **Dimension** node, double-click the **Subtypes** node.

   Content Server displays the “Subtypes for Asset Type: Dimension” form.
4. In the form, click the **Edit** (pencil) icon next to the **Locale** subtype.

   Content Server displays the “Edit Dimension Subtype: Locale” form.
5. In the **Sites** field, **Ctrl+click** the site in which you want to enable the “Locale” subtype.

   **Note**

   You must **Ctrl+click** the name of your site in order to keep the existing site selections intact; if you simply click on the site name, other selected sites (if any) will be deselected.
6. Click **Save**.

## Creating a Locale

To add a new locale to your site, create a “Dimension” asset of subtype “Locale” representing the desired locale, by performing the following steps:

**Tip**

If the locale you want to create designates a language that is already represented by a “Dimension” asset in another site in your CS system, share the existing “Dimension” asset representing that language to your current site instead to avoid redundancy.

**To create a locale**

1. In the button bar, click **New**.

2. In the list of asset types, click **New Dimension**.
3. Content Server displays the "New Dimension" form.
4. In the "New Dimension" form, do the following:
   a. In the **Name** field, enter a descriptive name for the locale. FatWire recommends using the following convention as a best practice:

      `xx_YY`

      where:
      - `xx` is the two-letter ISO 639-1 language code (for example, `fr` for French)
      - `YY` is the two-letter ISO country code (for example, `CA` for Canada)

      To complete the above example, the name `fr_CA` would denote Canadian French.
   b. In the **Description** field, enter a description of the language this locale represents.
   c. In the "Subtype" drop-down list, select **Locale**.
   d. Click **Save**.
5. Create and assign an icon that uniquely represents the locale in the Dash interface. The icon will appear in the list of returned assets for each asset assigned this locale whenever a user performs a search or runs a tag. Do the following:
   a. Create a GIF image no larger than 20x20 pixels representing the locale (for example, the flag of the corresponding country).
   b. Name the file *`locale`*`.gif`. The file name determines the locale for which the icon will be displayed. Use the exact name you assigned to the locale in step 4a above. The name is case sensitive.
   a. Place the file in the following directory:

      `<cs_app_dir>/images/icons`

      where `<cs_app_dir>` is the directory of the deployed CS application on your application server.
6. Restart your application server for the icons to appear in the Dash interface.

## Sharing a Locale to Another Site

To share a locale to another site, you must share the corresponding "Dimension" asset.

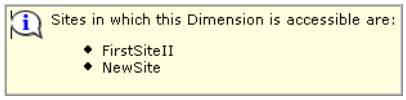
### To share a locale to another site

1. Log in to the Content Server interface and select the site containing the "Dimension" assets for the locales you want to share.
2. Find the desired "Dimension" asset and open its "Inspect" form:
   a. In the button bar, click **Search**.
   b. In the list of asset types, click **Find Dimension**.
   c. Enter the desired search criteria (if any), and click **Search**.
   d. In the list of search results, navigate to the desired asset and click its name.

      Content Server opens the asset in the "Inspect" form.
3. In the action bar, select **Share Dimension**.

Content Server displays the “Share Dimension” form.

4. In the “Share Dimension” form, select the check boxes next to the sites to which you want to share the “Dimension” asset. (To share the asset to all sites on your CS system, select the “All Sites” check box.)
5. Click **Save Changes**.
6. Content Server displays a message confirming the asset is now available in the sites you selected.




## Creating and Configuring a Dimension Set

### To create and configure a new dimension set

1. Add the “Dimension” assets (locales) you want to include in the dimension set to your Active List by doing the following:
   a. In the button bar, click **Search**.
   b. In the “Search” form, click **Find Dimension**.
   c. Enter the desired search criteria (if any) and click **Search**.
   d. In the list of search results, navigate to the desired **Dimension** assets and select their check boxes.
   e. Click **Add to My Active List**.
2. Create and configure the dimension set by doing the following:
   a. In the button bar, click **New**.
   b. In the “New” asset list, click **New DimensionSet**.

      Content Server displays the “New DimensionSet” form.
   c. In the **Name** field, enter a descriptive name for the dimension set.
   d. In the tree, select the **Active List** tab.
   e. In the **Active List** tab, select a locale you want to add to the dimension set and click **Add Selected Items**. Repeat this step for each additional locale you want to add.
   f. In the **Dimension Filter Class** field, select the desired locale filter type. The **Advanced** option allows you to specify a custom filter class.

      For more information on locale filter types, see “Working with Locale Filtering,” on page 647.
   g. (Optional) If you selected **Advanced** in step 2, enter the name of the custom filter class into the text box that appears.
   h. When you are finished, click **Save Changes**.
   i. (Optional) If you selected the **Hierarchical** filter in step 2, complete the steps in “Configuring the Fallback Hierarchy of the Hierarchical Filter,” on page 661 to configure the filter’s fallback hierarchy.

## Sharing a Dimension Set to Another Site

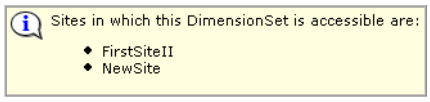
To share a dimension set to another site, you must share the corresponding "DimensionSet" asset.

### To share a dimension set to another site

1. Log in to Content Server and select the site containing the "DimensionSet" asset you want to share.
2. Find the desired "DimensionSet" asset and open its "Inspect" form:
   a. In the button bar, click **Search**.
   b. In the list of asset types, click **Find DimensionSet**.
   c. Enter the desired search criteria (if any), and click **Search**.
   d. In the list of search results, navigate to the desired asset and click its name.

      Content Server opens the asset in the "Inspect" form.
3. In the action bar, select **Share DimensionSet**.

   Content Server displays the "Share DimensionSet" form.
4. In the "Share DimensionSet" form, select the check boxes next to the sites to which you want to share the "DimensionSet" asset. (To share the asset to all sites on your CS system, select the "All Sites" check box.)
5. Click **Save Changes**.
6. Content Server displays a message confirming the asset is now available in the sites you selected.




## Configuring a Locale Filter

Usually, you configure the locale filter when you create the dimension set for your site. To make changes to the locale filter configuration in an existing dimension set, do the following:

### To configure a locale filter

1. Find the dimension set whose locale filter you want to configure and open in the "Inspect" form:
   a. In the button bar, click **Search**.
   b. In the "Search" form, click **Find DimensionSet**.
   c. Enter the desired search criteria (if any) and click **Search**.
   d. In the list of search results, navigate to the desired asset and click its name.

      Content Server opens the asset in the "Inspect" form.
2. In the **Dimension Filter Class** field, select the radio button next to the desired filter type. The **Advanced** option allows you to specify a custom filter class.

   For more information on locale filters, see "Working with Locale Filtering," on page 647.

3. (Optional) If you selected **Advanced** in step 2, enter the name of the custom filter class into the text field that appears.
4. Click **Save Changes**.
5. (Optional) If you selected the **Hierarchical** filter in step 2, complete the steps in "Configuring the Fallback Hierarchy of the Hierarchical Filter," on page 661 to configure the filter's fallback hierarchy.

## Configuring the Fallback Hierarchy of the Hierarchical Filter

If you selected the Hierarchical (Fallback) locale filter when configuring your dimension set, perform the following steps to configure the filter's fallback hierarchy:

**To configure the fallback hierarchy of the Hierarchical locale filter**

1. If you plan to add new locales or rearrange existing locales in the fallback hierarchy, add the "Dimension" assets representing the locales to be included in the hierarchy to your Active List by doing the following:
   a. In the button bar, click **Search**.
   b. In the "Search" form, click **Find Dimension**.
   c. Enter the desired search criteria (if any) and click **Search**.
   d. In the list of search results, navigate to the desired "Dimension" assets and select their check boxes.
   e. Click **Add to My Active List**.
2. Find and open in the "Inspect" form the dimension set that contains the Hierarchical filter you want to configure:
   a. In the button bar, click **Search**.
   b. In the "Search" form, click **Find DimensionSet**.
   c. Enter the desired search criteria (if any) and click **Search**.
   d. In the list of search results, navigate to the desired "DimensionSet" asset and click its hyperlinked name.

      Content Server displays the "DimensionSet" asset in the "Inspect" form.
3. Configure the fallback hierarchy:

   **Note**

   - A locale can only appear in the fallback hierarchy once.
   - To delete a locale from the hierarchy, click the **Delete** (trash can) icon next to the locale node.
   - To change the position of a locale in the hierarchy, delete it, then add it under the desired parent node.

   a. In the **Dimension Filter Class** field, click **Configure Locale Hierarchy**.

      Content Server displays the "Configure Locale Hierarchy" form.
   b. In the "Configure Locale Hierarchy" form, click **Edit**.

      Content Server displays an editable version of the form.
   c. In the tree, select the **Active List** tab.
   d. (Optional) If the hierarchy is empty, select in the **Active List** tab the locale you want to designate as the top node of the fallback hierarchy, then click **Add Selected Items**.
   e. Select a parent node for the locale you want to add to the hierarchy.

If you are building a hierarchy from scratch, your only choice will be the top-level node you added in step d.

When building your hierarchy, keep in mind the direction in which the fallback process occurs (from most-specific to least-specific; that is, towards the root node of the tree).

**f.** In the **Active List** tab, select the locale you want to appear under the parent node you selected in step e, then click **Add Selected Items**.

**g.** Repeat steps e and f for each additional locale you want to add to the hierarchy.

**h.** When your fallback hierarchy is complete, click **Save Changes**.

## Bulk-Assigning a Default Locale to Assets in a Site

If you are converting a monolingual site to a multilingual site, you must assign a default locale to all assets in the site. The fastest way to accomplish this is to execute an element that assigns the default locale to the assets.

**Note**

For your convenience, sample element code for this procedure is provided in the section, "Sample Element Code for Bulk-Assigning a Default Locale," on page 663. The sample code is intended as an example, and will have to be customized for your site.

### To bulk-assign a default locale to assets in a site

1. Create a CSElement asset to hold the element code that will assign a default locale to your assets. For instructions, see the *Content Server Developer's Guide*.
2. Create a SiteEntry asset that references the CSElement asset you created in step 1. For instructions, see the *Content Server Developer's Guide*.
3. Call the SiteEntry asset you created in step 2 in a URL, as follows:

   ```
   http://<host>:<port>/<context>/ContentServer?pagename=
      <siteentry_name>
   ```

   where:

   - `<host>` is the host of your CS system
   - `<port>` is the port number on which CS is listening for connections
   - `<context>` is the application context root assigned to the CS application.
   - `<siteentry_name>` is the name of the SiteEntry asset you created in step 2

   When the element code completes execution, check to make sure that your assets have the desired locale assigned. If not, check the element code for possible errors.

## Sample Element Code for Bulk-Assigning a Default Locale

This section contains sample element code written for the FirstSite II sample site. The code does the following:

1. Creates a “Dimension” asset named `en_US` to represent your default locale designation within the site (US English in this example).
2. Assigns this default locale to all “Page” assets within the site.

**Note**

The code in this section is provided as an example. If you decide to use it, be sure to customize it for your site. Test the code before deploying it; no error checking is included in this example.

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld"%>
<%@ taglib prefix="asset" uri="futuretense_cs/asset.tld"%>
<%@ taglib prefix="ics" uri="futuretense_cs/ics.tld"%>
<%@ taglib prefix="render" uri="futuretense_cs/render.tld"%>
<%@ taglib prefix="user" uri="futuretense_cs/user.tld"%>
<cs:ftcs>

<%-- Record dependencies for the SiteEntry and the CSElement --%>
<ics:if condition='<%=ics.GetVar("seid")!=null%>'>
<ics:then>
<render:logdep cid='<%=ics.GetVar("seid")%>' c="SiteEntry"/>
</ics:then>
</ics:if>

<ics:if condition='<%=ics.GetVar("eid")!=null%>'>
<ics:then>
<render:logdep cid='<%=ics.GetVar("eid")%>' c="CSElement"/>
</ics:then>
</ics:if>

<%-- log in as firstsite--%>
<user:login username="firstsite" password="firstsite"/>
<%-- create the Dimension asset (this can be done manually) --%>
<asset:create name="en_US" type="Dimension"/>
<asset:setsubtype name="en_US" value="Locale"/>
<asset:set name="en_US" field="name" value='en_US'/>
<asset:set name="en_US" field="description" value='US English'/>
<%-- enter your site's pubid below --%>
<ics:setvar name="primarypubid" value="1112198287026"/>
<asset:save name="en_US"/>
```

```
<%-- look up the id of the Dimension asset you just created --%>
<asset:get name="en_US" field="id" output="en_US.id"/>

<%-- get a list of all Content_C assets in the site, and assign a
   dimension to each of them --%>
<asset:list type="Content_C" list="allContentAssets"
   pubid="1112198287026"/>
<ics:listloop listname="allContentAssets">
<ics:listget listname="allContentAssets" fieldname="id"
   output="id"/>
<asset:load type="Content_C" objectid='<%=ics.GetVar("id")%>'
   name="tempName" editable="true"/>
<asset:adddimension name="tempName"
   dimensionid='<%=ics.GetVar("en_US.id")%>'/>
<asset:save name="tempName"/>
</ics:listloop>

</cs:ftcs>
```

Chapter 30

# Setting Up Flash Content Management

This chapter explains how to set up the data model for storing and managing Flash content in Content Server.

This chapter contains the following sections:


- Overview
- Prerequisites for Setting Up Flash Content Support
- Configuring Flash Content Support on a Site
- Sample Element Code for Rendering Flash Assets

# Overview

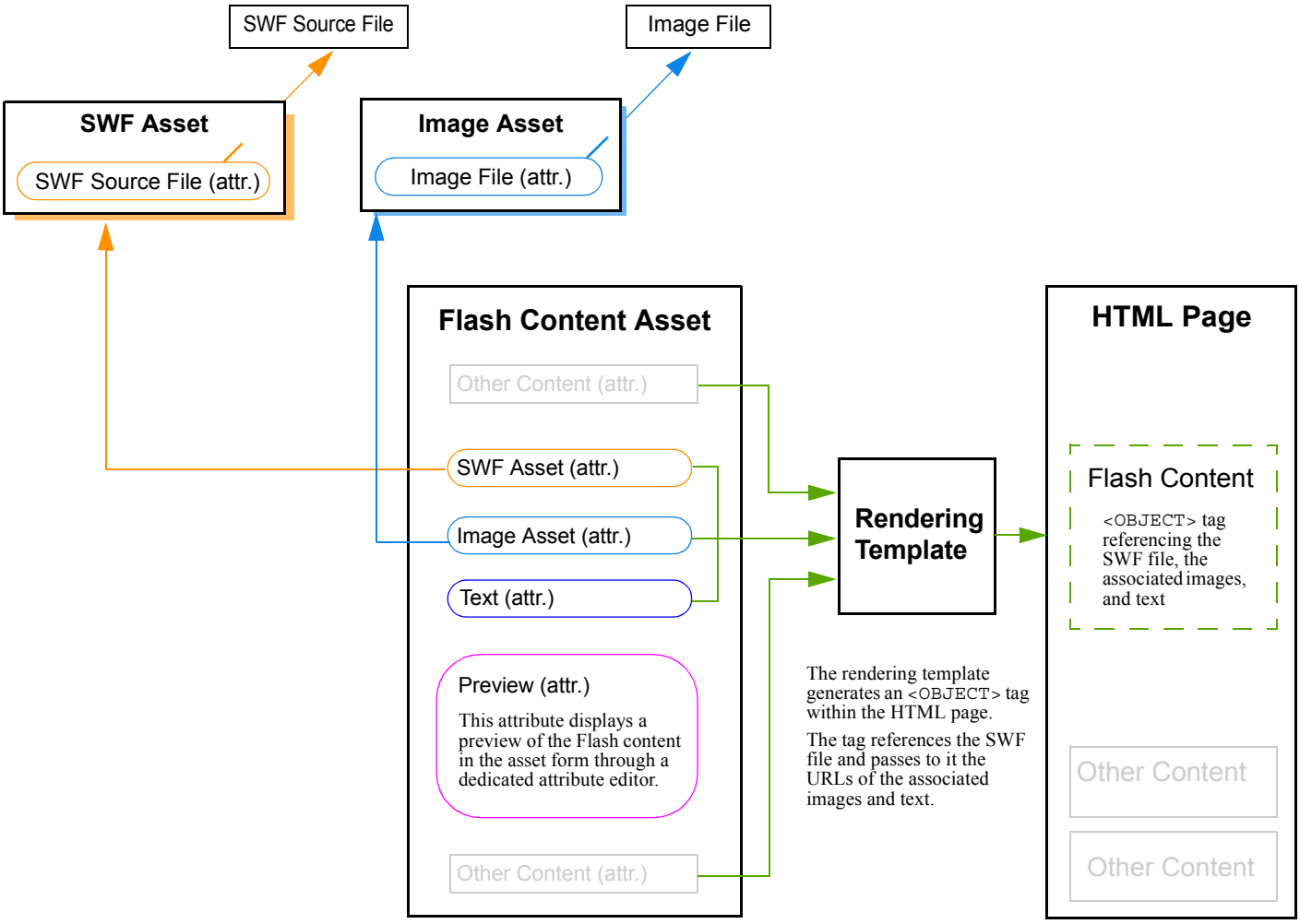
This chapter describes how to add Flash support to a site. The included procedures are examples based on the FirstSite II sample site. Use them as a reference when configuring your site.

## The Flash Management Model

Content Server can be set up to allow users to create, store, and publish Flash content. Users compose the Flash content by attaching SWF and image assets, and adding text to a Flash content asset, then publishing the asset to the online site. (The Flash content asset is a flex asset; the Flash management model does not support basic assets.) At render time, the URLs to the images and text are passed to the SWF file via an `<OBJECT>` tag generated by the rendering template.

As a developer, your job is to support this process by enabling a flex family (or families) on a site to recognize Flash content. At the very minimum, you need to create the data model components shown in Figure 14. Each component is described in "Configuring Flash Support," on page 667.

**Figure 14:** Flash Management Model



## Configuring Flash Support

You can implement Flash content on your site either by adding Flash support to an existing flex family or by creating a new flex family to which you will add Flash support. In either case, the target flex family must contain or support the following components, which you will create using the procedures in this chapter (the components are shown in Figure 14, on page 666):

- A **SWF asset type** whose flex definition contains a **SWF source file attribute**.

  The purpose of a SWF asset is to serve as a container for a SWF file. The SWF source file attribute allows for attachment of the **SWF source file** to the asset. This asset type can (but is not required to) be a member of the target flex family (the instructions in the next sections assume it is).

  **Note**

  A SWF source file is a Flash file that is parameterized to accept images and text by reference. As a result, the SWF source file is dynamic; that is, its content depends on which images and text are being referenced.

- An **image asset type** (whose flex definition contains an **image file attribute**).

  We assume that you have already configured an asset type from which content providers can create image assets. This asset type can (but is not required to) be a member of the target flex family (the instructions in the next sections assume it is not).

- A **Flash content asset type**, within the target flex family, whose flex definition contains the following attributes:
  - A **SWF asset attribute**. This attribute will allow content providers to associate SWF assets with Flash content assets.
  - An **image asset attribute**. This attribute will allow content providers to select images for embedding in the SWF source file and associate the images with a Flash content asset.

    **Note**

    Before creating this attribute, you will create an instance of the Image Picker attribute editor that will allow content providers to visually select the desired images.

  - A **text attribute**. This attribute will allow content providers to enter text to be embedded in the SWF file.
  - A **preview attribute**. This attribute will display the Flash content (i.e., the combination of the SWF file, images, and text) directly in the asset form.

    **Note**

    Before creating this attribute, you will create an instance of the `RENDERFLASH` attribute editor, which provides the mechanism for displaying the preview.

- **A rendering template** that will render Flash content assets on the online site by generating the necessary `<OBJECT>` tags.

## How Flash Content Is Rendered

At render time, the rendering template generates an `<OBJECT>` tag referencing the `SWF` file, the associated images, and text. This tag is then embedded in the HTML code of the target page. When a visitor requests the page, the browser renders the `SWF` file and the URLs of the images and text are passed to the `SWF` file as parameters (`FlashVars`). See "Sample Element Code for Rendering Flash Assets," on page 674 for sample element and `<OBJECT>` tag code that accomplishes this.

# Prerequisites for Setting Up Flash Content Support

Before you configure Flash content support on your site, you must satisfy the following prerequisites:

1. Decide on the nature of the Flash content. Typically, site designers specify the nature of the SWF source files and the image file so that they form complete pieces of content when combined.
2. Create the appropriate SWF source file(s).

   **Note**

   The SWF file must be coded to accept images and corresponding text (strings) by reference. At render time, the CS template rendering the Flash content generates an `<OBJECT>` tag that references the SWF file and passes to it the locations of the associated images and text. See "The Generated <OBJECT>/<EMBED> Tag," on page 679 for sample code for this tag.

3. Prepare the appropriate images:
   a. Create the appropriate images.
   b. Store the images as image assets in your Content Server system.
   c. (Optional) Categorize the images.

      Categorizing your images allows you to group images into categories (such as Audio, Video, or Electronics). You can then decide which categories the content providers will be able to choose images from when they compose Flash content.

      In other words, you can restrict the images available to the content providers to only those images that are relevant to the SWF files content providers will use. For instructions, see "Configuring the Image Picker," on page 403.

4. Create the target flex family, if you do not already have one, as described in "Creating a Flex Asset Family," on page 312. The table below provides example names and descriptions for each flex family member. The names you enter become names of the corresponding database tables, and are used in element code. The descriptions you enter appear in the user interface.

| Flex Asset Type | Example Parameters |
|---|---|
| **Flex Attribute** | Example name: **Flash_A**<br>Example description: **Flash Attribute** |
| **Flex Parent Definition** | Example name: **Flash_PD**<br>Example description: **Flash Parent Definition** |
| **Flex Definition** | Example name: **Flash_D**<br>Example description: **Flash Definition** |
| **Flex Parent** | Example name: **Flash_P**<br>Example description: **Flash Parent** |
| **Flex Asset** | Example name: **Flash_C**<br>Example description: **Flash Content Asset** |
| **Flex Filter** | Example name: **Flash_F**<br>Example description: **Flash Filter** |

5. Enable the flex family in the target site. For instructions, see "Step 3: Enable the New Flex Asset Types," on page 316.

**Note**

When you enable the flex family, instruct Content Server to create the corresponding "New" and "Search" start menu items.

# Configuring Flash Content Support on a Site

**Note**

Names of assets, asset types, and attributes, as well as parameter values shown in this section, are examples. In your installation, use values that conform to your organization's data model.

## I. Add the SWF Asset Type to the Target Flex Family

In this step, you will add the SWF asset type to the target flex family and enable it in the target site.

1. Add the SWF asset type to the target flex family. For instructions, see "Step 2: (Optional) Create Additional Flex Family Members," on page 315.
   - Example name (appears in the CS database): **SWFAsset_C**
   - Example description (appears in the UI): **SWF Asset**
2. Enable the new flex asset type in the target site. For instructions, see "Step 3: Enable the New Flex Asset Types," on page 316.

**Note**

When you enable the asset type, instruct Content Server to create the corresponding "New" and "Search" start menu items.

## II. Configure the SWF Asset Type

In this step, you will create the attributes required by the SWF asset type and assign them to the SWF asset flex definition.

1. Create the attribute for associating SWF files with SWF assets, using the following values. For instructions, see "Creating Flex Attributes of Type Blob (Upload Field)," on page 319.
   - Example name (appears in the UI): **SWFSourceFile**
   - Value Type: **blob**
   - Number of values: **single**

   Leave all other options at their default values.
2. Create the flex definition for the SWF asset using the following values.
   For instructions, see "Step 7: Create Flex Definition Assets," on page 324.
   - Example name (appears in the UI): **SWF Asset**
   - Attributes: **SWFSourceFile**

   Leave all other options at their default values.

## III. Create the Required Attribute Editor Instances

In this step you will create the required instances of the Image Picker and `RENDERFLASH` attribute editors as described below. You will use these instances with the attributes you will create in Step IV.

**Note**

When creating the attribute editor instances, make sure you provide the correct parameter values within their XML definitions. For a detailed list of parameters, see the respective sections noted below.

1. The Image Picker instance allows content providers to visually browse and select images to be embedded in the Flash content. Create it as follows:
   a. See "Configuring the Image Picker," on page 403 for the list of parameters whose values you will have to provide when creating the instance.
   b. Create the Image Picker instance by following the instructions in "Creating Attribute Editors," on page 365. (This instance is named **ImagePickerFlash** in our example.)
2. The `RENDERFLASH` attribute editor instance allows content providers to preview the Flash content directly in the Flash content asset form. Create it as follows:
   a. See "Configuring the RENDERFLASH Attribute Editor," on page 405 for the list of parameters whose values you will have to provide when creating the instance.
   b. Create the `RENDERFLASH` attribute editor instance by following the instructions in "Creating Attribute Editors," on page 365. (This instance is named **FlashPreview** in our example.)

## IV. Configure the Flash Content Asset Type

In this step, you will create the attributes required by the Flash content asset type and assign them to the Flash content asset flex definition (if necessary, you will also create the definition).

1. Create the following flex attributes, using the parameter values listed for each attribute. For instructions, see "Step 4: Create Flex Attributes," on page 317.
   a. Create the **SWF asset attribute**. This attribute will be used for associating SWF assets with Flash content assets:
      - Example name (appears in the UI): **SWFAsset**
      - Value Type: **asset**
      - Asset Type: **SWF Asset**
      - Number of values: **single**
      - Attribute editor: **PickAsset**

      Leave all other options at their default values.

b. Create the **image asset attribute**. This attribute will be used for associating image assets with Flash content assets.
   - Example name (appears in the UI): **FlashImages**
   - Value Type: **asset**
   - Asset Type: **Media**
   - Number of values: **multiple (ordered)**
   - Attribute Editor: **ImagePickerFlash** (you created this attribute editor in step 1 on page 671)

   Leave all other options at their default values.

c. Create the **text attribute**. This attribute will hold the text that will be passed to the SWF file at render time:
   - Example name (appears in the UI): **FlashText**
   - Value Type: **string**
   - Number of values: **multiple (ordered)**

   Leave all other options at their default values.

d. Create the **preview attribute**. This attribute will display a preview of the Flash content (that is, the SWF file with the images and text added) directly in the asset form:
   - Example name (appears in the UI): **FlashPreview**
   - Value Type: **string**
   - Number of values: **single**
   - Attribute editor: **FlashPreview** (you created this attribute editor in step 2 on page 671)

   Leave all other options at their default values.

2. Depending on whether you are adding Flash content support to an existing flex family, do one of the following:
   - If you are adding Flash content support to an existing flex family, edit the flex definition for the Flash content asset and add the attributes listed below by carrying out the procedure, "Step 7: Create Flex Definition Assets," starting with step 5 on page 325.

     Attributes (preserve the order shown): **SWFAsset, FlashImages, FlashText, FlashPreview**
   - If you are adding Flash content support to a newly created flex family, create the flex definition for the Flash content asset using the parameter values shown below. For instructions, see "Step 7: Create Flex Definition Assets," on page 324.
     - Example name: **Flash Asset**
     - Attributes (preserve the order shown): **SWFAsset, FlashImages, FlashText, FlashPreview**

     Leave all other options at their default values.

## V. Create the Flash Rendering Template

Create a "Template" asset (named **RenderFlashAsset** in our example) that will render Flash content on your online site. For instructions, see the following:

- Chapter 23, "Creating Template, CSElement, and SiteEntry Assets"
- Chapter 27, "Coding Elements for Templates and CSElements"

For sample element code, see "Sample Element Code for Rendering Flash Assets," on page 674.

## VI. Test Your Configuration

1. Create a sample SWF asset and assign an appropriate SWF source file to it. For instructions on creating assets, see the *Content Server Advanced Interface User's Guide*.
2. Create a sample Flash content asset to make sure that your `RENDERFLASH` attribute editor instance has been configured properly. If no preview is displayed when creating the Flash content asset, check that you are passing the correct parameters to the `RENDERFLASH` attribute editor in its XML definition. For details, see "Configuring the RENDERFLASH Attribute Editor," on page 405.
3. Assign the Flash rendering template you created in "V. Create the Flash Rendering Template" to the sample Flash content asset and place the asset on a page. Preview the page to make sure it is rendered properly.

**Note**

The Flash rendering template generates an `<OBJECT>` tag (with a nested `<EMBED>` tag) within the page. The tag serves the following functions:

- References the SWF source file
- Passes the URLs of the images (stored as image assets) and text (stored within the Flash asset being rendered) to the SWF source file.

For more information, see "The Generated <OBJECT>/<EMBED> Tag," on page 679.

If rendering anomalies occur, review your data model and your element code, correct any problems you find, and preview the asset again.

# Sample Element Code for Rendering Flash Assets

Sample element code for rendering Flash assets on your online site (based on the FirstSite II sample site) is included below for your reference. Examine it to get an idea of how to display Flash assets on your site.

```
<%@ taglib prefix="cs" uri="futuretense_cs/ftcs1_0.tld"
%><%@ taglib prefix="asset" uri="futuretense_cs/asset.tld"
%><%@ taglib prefix="assetset" uri="futuretense_cs/assetset.tld"
%><%@ taglib prefix="commercecontext" uri="futuretense_cs/
commercecontext.tld"
%><%@ taglib prefix="ics" uri="futuretense_cs/ics.tld"
%><%@ taglib prefix="listobject" uri="futuretense_cs/
listobject.tld"
%><%@ taglib prefix="render" uri="futuretense_cs/render.tld"
%><%@ taglib prefix="siteplan" uri="futuretense_cs/siteplan.tld"
%><%@ taglib prefix="searchstate" uri="futuretense_cs/
searchstate.tld"
%><%@ taglib prefix="blobservice" uri="futuretense_cs/
blobservice.tld"
%><%@ taglib prefix="insite" uri="futuretense_cs/insite.tld"
%><%@ page import="COM.FutureTense.Interfaces.*,
                   COM.FutureTense.Util.ftMessage,
                   COM.FutureTense.Util.ftErrors,
                   java.net.URLEncoder"
%><cs:ftcs><%-- GM_C/renderflashasset

INPUT

OUTPUT

--%>

<%-- Record dependencies for the Template --%>
<ics:if
condition='<%=ics.GetVar("tid")!=null%>'><ics:then><render:logdep
cid='<%=ics.GetVar("tid")%>' c="Template"/></ics:then></ics:if>

<insite:editasset assettype='<%=ics.GetVar("c")%>'
assetid='<%=ics.GetVar("cid")%>' dmv="true" displayname="Edit
Flash"/>
<insite:createasset assettypes='<%=ics.GetVar("c")%>' dmv="true"
displayname="Create Flash"/>

<blobservice:getidcolumn varname="idcol"/>
<blobservice:gettablename varname="tablename"/>
<blobservice:geturlcolumn varname="urlcol"/>
```

```
<ics:setvar name="FFATTRTYPEOUT" value="Flash_A"/>
<ics:setvar name="FFATTRNAMEOUT" value="SWFSourceFile"/>
<ics:setvar name="FFTYPEOUT" value="SWFAsset_C"/>
<ics:setvar name="IFATTRTYPEOUT" value="Media_A"/>
<ics:setvar name="IFATTRNAMEOUT" value="FSII_ImageFile"/>
<ics:setvar name="IFTYPEOUT" value="Media_C"/>
<ics:setvar name="FAATTRTYPEOUT" value="Flash_A"/>
<ics:setvar name="FAFLASHATTRNAMEOUT" value="SWFAsset"/>
<ics:setvar name="FAIMAGEATTRNAMEOUT" value="FlashImages"/>
<ics:setvar name="FATEXTATTRNAMEOUT" value="FlashText"/><br/>
<p><br>
<img src="http://localhost:8080/cs/images/gmnav.jpg"><br>
<assetset:setasset name="flashasset" type='<%=ics.GetVar("c")%>'
id='<%=ics.GetVar("cid")%>' />

<assetset:getattributevalues name="flashasset"
typename='<%=ics.GetVar("FAATTRTYPEOUT")%>'
attribute='<%=ics.GetVar("FAFLASHATTRNAMEOUT")%>'
listvarname="flashlist"/>
<assetset:getattributevalues name="flashasset"
typename='<%=ics.GetVar("FAATTRTYPEOUT")%>'
attribute='<%=ics.GetVar("FAIMAGEATTRNAMEOUT")%>'
listvarname="imagelist"/>
<assetset:getattributevalues name="flashasset"
typename='<%=ics.GetVar("FAATTRTYPEOUT")%>'
attribute='<%=ics.GetVar("FATEXTATTRNAMEOUT")%>'
listvarname="txtlist"/>

<ics:listloop listname="flashlist"><ics:listget
listname="flashlist" fieldname="value" output="flashid"/></
ics:listloop>
<%String sImgVal =""; %>
<ics:listloop listname="imagelist">
<ics:listget listname="imagelist" fieldname="value"
output="imgid"/>
<assetset:setasset name="imagefile"
type='<%=ics.GetVar("IFTYPEOUT")%>' id='<%=ics.GetVar("imgid")%>'
/>
<assetset:getattributevalues name="imagefile"
typename='<%=ics.GetVar("IFATTRTYPEOUT")%>'
attribute='<%=ics.GetVar("IFATTRNAMEOUT")%>'
listvarname="imagefilelist"/>
<ics:listloop listname="imagefilelist">
<ics:listget listname="imagefilelist" fieldname="value"
output="imgfileid"/>
<render:getbloburl
      outstr="imgurl"
      blobtable="<%=ics.GetVar("tablename")%>"
      blobkey="<%=ics.GetVar("idcol")%>"
      blobwhere="<%=ics.GetVar("imgfileid")%>"
```

```
      blobcol="<%=ics.GetVar("urlcol")%>"
      scheme="http"
  authority="localhost:8080"
      >
</render:getbloburl>
<%
   String sImgURL="";
   //sImgURL=getBaseURL(request);
   //sImgURL="http://localhost:8080/cs/BlobServer?";
   sImgURL=request.getScheme()+"://
"+request.getServerName()+":"+request.getServerPort()+request.getC
ontextPath()+"/BlobServer?";
   sImgURL=addParam(sImgURL,"blobtable",ics.GetVar("tablename"));
   sImgURL=addParam(sImgURL,"blobkey",ics.GetVar("idcol"));
   sImgURL=addParam(sImgURL,"blobwhere",ics.GetVar("imgfileid"));
   sImgURL=addParam(sImgURL,"blobcol",ics.GetVar("urlcol"));
   sImgVal = sImgVal+sImgURL+"|";
%>
</ics:listloop>
</ics:listloop>
<% //out.println(sImgVal);%>
<%String sTxtVal =""; %>
<ics:listloop listname="txtlist">
<ics:listget listname="txtlist" fieldname="value" output="txtval"/
>
<%
 sTxtVal = sTxtVal+ics.GetVar("txtval")+"|";
%>
</ics:listloop>



<assetset:setasset name="flashfile"
type='<%=ics.GetVar("FFTYPEOUT")%>'
id='<%=ics.GetVar("flashid")%>' />
<assetset:getattributevalues name="flashfile"
typename='<%=ics.GetVar("FFATTRTYPEOUT")%>'
attribute='<%=ics.GetVar("FFATTRNAMEOUT")%>'
listvarname="flashfilelist"/>
<% String
sFlashURL=request.getServerName()+":"+request.getServerPort();%>


<ics:listloop listname="flashfilelist"><ics:listget
listname="flashfilelist" fieldname="value" output="flashfileid"/
></ics:listloop>


<render:getbloburl
      outstr="flashurl"
      blobtable="<%=ics.GetVar("tablename")%>"
```

```
      blobkey="<%=ics.GetVar("idcol")%>"
      blobwhere="<%=ics.GetVar("flashfileid")%>"
      blobcol="<%=ics.GetVar("urlcol")%>"
      satellite="false"
  scheme="http"
  authority="<%=sFlashURL%>"
      >
</render:getbloburl>

<OBJECT classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=6,0,0,0'
WIDTH='525'
HEIGHT='154'
id='fs1'
align='middle'>

<PARAM NAME='FlashVars'
VALUE='imagefiles=<%=sImgVal%>&imagelabels=<%=sTxtVal%>'>
<PARAM NAME='allowScriptAccess' VALUE='sameDomain'>
<PARAM NAME='movie' VALUE='<%=ics.GetVar("flashurl")%>'>
<PARAM NAME='quality' VALUE='high'>
<PARAM NAME='bgcolor' VALUE='#FFFFFF'>
<PARAM NAME='scale' VALUE='noborder'>

<EMBED src='<%=ics.GetVar("flashurl")%>'
quality='high'
bgcolor='#FFFFFF'
FlashVars='imagefiles=<%=sImgVal%>&imagelabels=<%=sTxtVal%>'
WIDTH='525'
HEIGHT='154'
NAME='fs1'
scale='noborder'
align='middle'
allowScriptAccess='sameDomain'
TYPE='application/x-shockwave-flash'
PLUGINSPAGE='http://www.macromedia.com/go/getflashplayer'>

</EMBED>

<%! private String
getBaseURL(javax.servlet.http.HttpServletRequest request) throws
Exception
{
String sReturn = "";
sReturn = sReturn + request.getRequestURI() + "?";
```

```
return sReturn;
}
%>

<%! private String addParam(String sURL, String key, String value)
throws Exception
{
String sReturn = sURL;
sReturn = sReturn + key + "=" + value + "%26";
return sReturn;
}
%>

</cs:ftcs>
```

## The Generated <OBJECT>/<EMBED> Tag

The Flash rendering template generates an `<OBJECT>` tag (containing an `<EMBED>` tag with the same information) referencing the images and supporting text that must be passed to the SWF file at render time.

An example tag (based on the FirstSite II sample site) is shown below for your reference. Use it to get an idea of how parameters (`imagefiles`, `imagelabels`) are passed by the Flash rendering template to the SWF file (`movie`). In our example, three images with three corresponding strings are passed to the SWF file referenced in the `movie` field.

```
<OBJECT classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=6,0,0,0'
WIDTH='525'
HEIGHT='154'
id='flash_1174718906041_flashrender'
align='middle'>
<PARAM NAME='FlashVars' VALUE='imagefiles=http://localhost:8080/
cs/
BlobServer?blobtable=MungoBlobs%26blobkey=id%26blobwhere=117471891
8434%26blobcol=urldata%26|http://localhost:8080/cs/
BlobServer?blobtable=MungoBlobs%26blobkey=id%26blobwhere=117471891
1540%26blobcol=urldata%26|http://localhost:8080/cs/
BlobServer?blobtable=MungoBlobs%26blobkey=id%26blobwhere=117471891
1540%26blobcol=urldata%26|&imagelabels=1st Cadillac|2nd
Cadillac|3rd Cadillac|'>
<PARAM NAME='allowScriptAccess' VALUE='sameDomain'>
<PARAM NAME='movie' VALUE='http://localhost:8080/cs/
BlobServer?blobcol=urldata&blobtable=MungoBlobs&blobkey=id&blobwhe
re=1174718911787'>

<PARAM NAME='quality' VALUE='high'>
<PARAM NAME='bgcolor' VALUE='#FFFFFF'>
<PARAM NAME='scale' VALUE='noborder'>

<EMBED src='http://localhost:8080/cs/
BlobServer?blobcol=urldata&blobtable=MungoBlobs&blobkey=id&blobwhe
re=1174718911787'
quality='high'
bgcolor='#FFFFFF'
FlashVars='imagefiles=http://localhost:8080/cs/
BlobServer?blobtable=MungoBlobs%26blobkey=id%26blobwhere=117471891
8434%26blobcol=urldata%26|http://localhost:8080/cs/
BlobServer?blobtable=MungoBlobs%26blobkey=id%26blobwhere=117471891
1540%26blobcol=urldata%26|http://localhost:8080/cs/
BlobServer?blobtable=MungoBlobs%26blobkey=id%26blobwhere=117471891
1540%26blobcol=urldata%26|&imagelabels=1st Cadillac|2nd
Cadillac|3rd Cadillac|'
WIDTH='525'
HEIGHT='154'
NAME='flash_1174718906041_flashrender'
scale='noborder'
```

```
align='middle'
allowScriptAccess='sameDomain'
TYPE='application/x-shockwave-flash'
PLUGINSPAGE='http://www.macromedia.com/go/getflashplayer'>
</EMBED>
</OBJECT>
```

Chapter 31

# User Management on the Delivery System

Content Server provides authentication functionality through the `USER` tags, user profile management through the `DIR` tags, and enforces security on database tables and rendered pages through access control lists (ACLs). You use these user management and security mechanisms to manage users and control visitor access on your distribution system and on your Content Server development and management systems.

This chapter contains the following sections:


- The Directory Services API
- Controlling Visitor Access to Your Online Sites
- Creating Login Forms
- Creating User Account Creation Forms
- Visitor Access in the Burlington Financial Sample Site

# The Directory Services API

The Directory Services API enables your Content Server system to connect to directory servers that contain authentication information, user information, and so on.

Content Server delivers three directory services plug-ins, one of which was installed when your Content Server systems were installed:

- The Content Server directory services plug-in, which uses the native Content Server user management tables; that is, the `SystemUsers` and `SystemUserAttrs` tables
- The LDAP plug-in, which actually supports any JNDI server
- The NT plug-in, which retrieves user credentials and login name from the NT directory but gets all other user information from the `SystemUserAttrs` table

The plug-in is installed during the installation of your Content Server systems and it is configured by setting properties in the `dir.ini` file. For information about configuring your user management setup, see the *Content Server Administrator's Guide*.

## Entries

A directory entry is a named object with assigned attributes, in particular, user and group type entries:

- A user type object has a distinguished name and a set of attributes such as commonname, username, password and email.
- A group type object, similar to a Content Server ACL, also has a distinguished name and a set of attributes.

Names reflect the hierarchy in which they are associated; to ensure portability across directory implementations, names should be treated as opaque strings.

## Hierarchies

Some directory databases organize entries using a hierarchical structure. With Content Server's directory services API, an entry's attributes and its place in the hierarchy are distinct. As a result, retrieving an entry's attributes does not yield information about its children.

Support for hierarchies depends on the underlying directory implementation; for example, LDAP directories support a hierarchical structure, while Content Server's native directory database does not support a hierarchical structure.

To ensure portability across directory implementations, your code should not assume support for hierarchical data.

Group hierarchies do not affect internal Content Server permissions.

## Groups

Content Server's directory services API does not enforce referential integrity. When you delete a user with the directory tags, your application must ensure that group memberships are also deleted, by first removing the user from the groups that he is associated with.

When a member is added to a group, the JNDI implementation always builds a fully distinguished name for the value of the `uniquemember` attribute, regardless of the name passed into the `addmember` tag.

## Directory Services Tags

Content Server delivers the DIR tag family, with both XML and JSP versions, that you can use to invoke the Directory Services API.

The DIR tags are as follows:

| Tag | Description |
|---|---|
| DIR.ADDATTRS<br>dir:addattrs | Adds attributes to an existing entry (which can be either a user or a group). |
| DIR.ADDGROUPMEMBER<br>dir:addgroupmember | Adds a member to a group (usually a user). |
| DIR.CHILDREN<br>dir:children | Retrieves the child entries for a specified parent in a list variable. |
| DIR.CREATE<br>dir:create | Creates a directory entry. |
| DIR.DELETE<br>dir:delete | Deletes a directory entry. |
| DIR.GETATTRS<br>dir:getattrs | Gets the attribute values for a specified entry in a list variable. |
| DIR.GROUPMEMBERS<br>dir:groupmembers | Lists the members of a specified group. |
| DIR.GROUPMEMBERSHIPS<br>dir:groupmemberships | Lists all the groups that an entry (either a group or a user) belongs to. |
| DIR.LISTUSERS<br>dir:listusers | Returns a list of all the users in the directory. |
| DIR.REMOVEATTRS<br>dir:removeattrs | Deletes an attribute value for an entry. |
| DIR.REMOVEGROUPMEMBER<br>dir:removegroupmember | Removes an entry from a group. |
| DIR.REPLACEATTRS<br>dir.replaceattrs | Replaces the value of an attribute for an entry (either a user or a group). |
| DIR.SEARCH<br>dir:search | Searches the directory for entries who match the specified search criteria. |

Regardless of whether the directory is implemented with LDAP, Content Server only, or any other directory server, the code you write with the DIR tags is very similar.

For more information about these tags, see the *Content Server Tag Reference*. For code samples, see "Directory Services Code Samples" on page 684.

## Directory Operations

Some of the Content Server Directory Services tags write information to the database. If your database administrators will be handling all of the web site's write operations, such as adding user information to the database, restrict use of the directory tags to

read-only operations. This policy avoids synchronization issues with third-party directory administration tools.

The read-only operations are presented in this section. Operations are performed using the credentials and read permissions of the currently authenticated user.

## Searching

Due to limitations in some directory servers, search is not allowed from the top organizational level. To avoid portability issues, always specify the context attribute on the `DIR.SEARCH` tag.

## Lookup

Looking up a user generally involves two steps:

1. Call `DIR.SEARCH` on the `userid` to get the entry name.
2. Call `DIR.GETATTRS` to get the attributes of the user in question.

## Listing Users

FatWire recommends that you use one of the following three methods to list users:

- For small user databases, use the `DIR.LISTUSERS` tag, which recursively lists all users under the `peopleParent` property. This tag is inefficient on large user databases.
- For large user databases, use the `DIR.CHILDREN` tag to walk the hierarchy. The `DIR.CHILDREN` tag is best used for group types and not for user types.
- For user databases with a flat hierarchy, narrow results with a search

## Directory Services Code Samples

The following JSP code sample illustrates some possible directory operations:

```
<%
String sMainTestUserName = "ContentServer";
String sMainTestUserPW="FutureTense";

String sPeopleParent = ics.GetProperty("peopleparent", "dir.ini",
true);
String sGroupParent = ics.GetProperty("groupparent", "dir.ini",
true);
String sUsername = ics.GetProperty("username", "dir.ini", true);
String sCommonName = ics.GetProperty("cn", "dir.ini", true);
IList mylist;
%>

<user:su username='<%=sMainTestUserName%>'
password='<%=sMainTestUserPW%>'>

<H2>List All Users</H2>

<ics:clearerrno/>
<dir:listusers list='mylist'/>
<br>
```

```
<b>dir:listusers errno: <ics:getvar name='errno'/></b>
<ics:listloop listname='mylist'>
   <br><ics:listget listname='mylist' fieldname='NAME'/>
</ics:listloop>

<H2>Look Up the ContentServer User by Username</H2>

<ics:clearerrno/>
<dir:search list='mylist' context='<%=sPeopleParent%>'>
   <dir:argument name='<%=sUsername%>' value='ContentServer'/>
</dir:search>
<br><b>dir:search errno: <ics:getvar name='errno'/></b>

<%
   mylist = ics.GetList("mylist");
   if(mylist.numRows() != 1) {
         out.print("<br>Error finding entry.");
   }
   mylist.moveTo(1);
   ics.SetVar("ContentServerDn", mylist.getValue("NAME"));
%>

<H2>Show ContentServer Attributes</H2>

<ics:clearerrno/>
<dir:getattrs list='mylist'
   name='<%=ics.GetVar("ContentServerDn")%>'/>
<br><b>dir:getattrs errno: <ics:getvar name='errno'/></b>
<ics:listloop listname='mylist'>
   <br>
   <ics:listget listname='mylist' fieldname='NAME'/>=
   <ics:listget listname='mylist' fieldname='VALUE'/>
</ics:listloop>

<H2>Show Group Memberships for ContentServer</H2>

<ics:clearerrno/>
<dir:groupmemberships name='<%=ics.GetVar("ContentServerDn")%>'
   list='mylist'/>
<br><b>dir:groupmemberships errno: <ics:getvar name='errno'/></b>
<ics:listloop listname='mylist'>
<br>
   <ics:listget listname='mylist' fieldname='NAME'/>
</ics:listloop>

<H2>Lookup the SiteGod Group by CommonName</H2>

<ics:clearerrno/>
<dir:search list='mylist' context='<%=sGroupParent%>'>
   <dir:argument name='<%=sCommonName%>' value='SiteGod'/>
</dir:search>
<br><b>dir:search errno: <ics:getvar name='errno'/></b>
```

```
<%
   mylist = ics.GetList("mylist");
   if(mylist.numRows() != 1) {
         out.print("<br>Error finding entry.");
   }
   mylist.moveTo(1);
   ics.SetVar("SiteGodDn", mylist.getValue("NAME"));
%>

<H2>Show SiteGod Attributes</H2>

<ics:clearerrno/>
<dir:getattrs list='mylist' name='<%=ics.GetVar("SiteGodDn")%>'/>
<br>
<b>dir:getattrs errno: <ics:getvar name='errno'/></b>
<ics:listloop listname='mylist'>
   <br>
   <ics:listget listname='mylist' fieldname='NAME'/>=
   <ics:listget listname='mylist' fieldname='VALUE'/>
</ics:listloop>

<H2>Show SiteGod Group Members</H2>

<ics:clearerrno/>
<dir:groupmembers name='<%=ics.GetVar("SiteGodDn")%>'
list='mylist2'/>
<br>
<b>dir:groupmembers errno: <ics:getvar name='errno'/></b>
<ics:listloop listname='mylist2'>
   <br>
   <ics:listget listname='mylist2' fieldname='NAME'/>
</ics:listloop>

<H2>Children of groupparent </H2>

<ics:clearerrno/>
<dir:children name='<%=sGroupParent%>' list='mylist'/>
<br>
<b>dir:children errno: <ics:getvar name='errno'/></b>
<ics:listloop listname='mylist'>
   <br>
   <ics:listget listname='mylist' fieldname='NAME'/>
</ics:listloop>

</user:su>
```

## Error Handling

Any of the directory tags can cause a range of directory errors to be set. See the *Content Server Tag Reference* for a comprehensive list of directory services error messages.

Your directory services code should handle every one of the error codes listed for a given tag call. This is necessary to support the J2EE JNDI interface.

## Troubleshooting Directory Services Applications

The first step in troubleshooting directory services applications is to check the error log (`futuretense.txt`).

You enable directory services logging by setting the `log.filterLevel` property (found in the `logging.ini` property file). There are seven levels of error messages that you can view:

- `fatal`, which logs fatal level messages
- `severe`, which logs severe and fatal level messages
- `error`, which logs error and fatal level messages
- `warning`, which logs warning and fatal level messages
- `info`, which logs warning, error, severe, and fatal level messages
- trace, which logs trace messages
- detail, which logs all types of messages

During troubleshooting, `trace` is the most verbose setting, and as a result, has the highest performance impact.

Directory services log entries use the following format:

```
[<timestamp>][Directory-<severity>-<errno>]
   [<class>:<method>][<message>][<session id>]
```

For example:

```
[Jan 17, 2002 1:49:44 PM][Directory-T]
   [BaseFactory:instantiateImplementation(ICS,String,Class[],
   Object[])][Instantiating:com.openmarket.directory.common.Factor
   y]
   [PEccxyF1Ueh7zYvjNgg4D6bqZzf0llfWMaiBimIN9H1Z9KomDcPy]
```

The previous message is a trace (T), and thus has no associated `errno` value.

For information about error logging, see Chapter 10, “Error Logging and Debugging.”

A common problem for LDAP implementations is incorrectly specified permissions on the directory server. If the error log indicates a permission problem, ensure that the authenticated user has permissions to execute the requested operation by checking the permission settings on the directory server. For iPlanet, this entails checking the groups to which the user belongs, and checking the LDAP ACLs associated with those groups. Try logging into the directory server directly (outside of Content Server) and performing the same action to ensure that permissions are correctly set.

After checking the log and permissions, you can often resolve a configuration error by examining the property files.

For property descriptions and values, see the *Content Server Property Files Reference*.

# Controlling Visitor Access to Your Online Sites

Content Server manages users through access control lists (ACLs). By using ACLs, you can restrict access to tables in the Content Server database and the rendered pages served on your sites by Content Server.

If you design an online site where visitors log in with user names and passwords, you can associate registered visitors with one or more ACLs.

When a visitor first visits a site, Content Server creates a session and implicitly logs in the visitor as the standard default user, **DefaultReader**. The identity of a visitor is updated (and any associated ACLs go into effect) when a `USER.LOGIN` command is used and the visitor is authenticated against a password.

## ACL Tags

Content Server provides a set of access control list tags (both XML and JSP versions) that you can use to create ACLs. You can use either the Content Server interface on the delivery system or the Content Server ACL tags to create the ACLs that you need for your visitor accounts on your delivery system.

The following table lists the ACL tags:

| Tag | Description |
|---|---|
| `ACL.CREATE`<br>`acl:create` | Creates an ACL |
| `ACL.DELETE`<br>`acl:delete` | Deletes an ACL |
| `ACL.GATHER`<br>`acl:delete` | Gathers fields into an ACL |
| `ACL.GET`<br>`acl:get` | Copies a field from an ACL |
| `ACL.LIST`<br>`acl:list` | Retrieves a list of ACLs |
| `ACL.LOAD`<br>`acl:load` | Loads an ACL |
| `ACL.SAVE`<br>`acl:save` | Saves an ACL |
| `ACL.SCATTER`<br>`acl:scatter` | Scatters a field from an ACL |
| `ACL.SET`<br>`acl:set` | Sets a field in an ACL |

For more information:

- ACL tags—see the *Content Server Tag Reference*.
- ACLs in general—see the *Content Server Administrator's Guide*.

## User Tags

Content Server also provides the following `USER` tags (both XML and JSP versions) that you use on pages that log users in and out.

The `USER` tags are as follows:

| Tag | Description |
|---|---|
| `USER.LOGIN`<br>`user:login` | Logs a user in. |
| `USER.LOGOUT`<br>`user:logout` | Logs a user out. |
| `USER.SU`<br>`user:su` | Logs the user in as a specific user in order to perform an operation such as creating an account or edit a user profile. |

For more information about these tags, see the *Content Server Tag Reference*.

## Content Server and Encryption

Content Server includes a default key for encrypting passwords and other sensitive information. You can specify your own encryption key by using the `Utilities` class `encryptString` method. See the *Content Server Javadoc* for information about Java methods that deal with encryption.

Content Server also supports Secure Sockets Layer (SSL), which allows encryption of information going to and from your web servers. For more information about Content Server and SSL, see the *Content Server Administrator's Guide*.

# Creating Login Forms

This section provides simple code samples that illustrate how to code login forms that prompt a visitor to log in and then authenticate the user name and password.

This example presents code from the following elements:

- `PromptForLogin`, an XML element that displays a form that requests a username and password
- `Login`, an XML element that authenticates the username and password combination passed to it from the `PromptForLogin` element

## Prompt for Login (PromptForLogin.xml)

The `PromptForLogin` element displays a form that asks a visitor to enter two pieces of information: username and password.

The code that creates the form follows:

```
<DIV ALIGN="center">
<FORM ACTION="ContentServer" METHOD="post">
<INPUT TYPE="hidden" NAME="pagename" VALUE="CSGuide/Security/
Login"/>
```

```
<TABLE CELLPADDING="5" CELLSPACING="5">

<TR>
<TH ALIGN="right">Name</TH>
<TD><INPUT TYPE="text" NAME="username" SIZE="16"/></TD>
</TR>

<TR>
<TH ALIGN="right">Password</TH>
<TD><INPUT TYPE="password" NAME="password" SIZE="16"/></TD>
</TR>

<TR>
<TD> </TD>
<TD ALIGN="center"><INPUT TYPE="submit" NAME="doit" VALUE="Login"/
></TD>
</TR>

</TABLE>

</FORM>
</DIV>
```

The visitor fills in the form and clicks the **Submit** button. The information gathered in the form and the page name of the `Login` page (see the first `input type` statement, above) is sent to the browser. The browser sends the page name to Content Server, Content Server looks it up in the `SiteCatalog` table and then invokes that page entry's root element.

## Root Element for the Login Page

There can only be one root element for a Content Server page (that is, an entry in the `SiteCatalog` table). The root element for the `Login page` is the `Login.xml` element.

The `Login` element attempts to log the visitor in and then to authenticate the visitor by using the `USERISMEMBER` tag to determine whether the visitor has any of the required ACLs.

This element does the following:

- Logs the visitor in with the `USER.LOGIN` tag and checks to see if there was an error.
- Sets a variable list that holds a list of ACLs to compare the visitor's credentials against.
- Checks the visitor's ACLs assignments against the list variable with the `USERISMEMBER` tag

The following sample code authenticates the visitor:

```
<SETVAR NAME="errno" VALUE="0"/>

<USER.LOGIN USERNAME="Variables.username"
PASSWORD="Variables.password"/>

<IF COND="Variables.errno=0">
      <THEN>
```

```
      <H3>Welcome <CSVAR NAME="Variables.username"/></H3>

<!-- Next, create a variable named “aclToCheck” and set it to hold
the ACLs that the visitor needs to progress any further on the
site. This variable can be set to one ACL or to a comma-separated
list of ACLs, as in this example-->

   <SETVAR NAME="aclToCheck" VALUE="ContentCustomer,SiteGod"/>
   <SETVAR NAME="errno" VALUE="0"/>

   <USERISMEMBER GROUP="Variables.aclToCheck"/>
   <IF COND="Variables.errno=1">
         <THEN>
         <P>You are a member of the at least one of the following
   acls:
         <CSVAR NAME="Variables.aclToCheck"/></P>
         </THEN>
       <ELSE>
         <P>Sorry, you are not a member of any of the following
acls:
         <CSVAR NAME="Variables.aclToCheck"/></P>
       </ELSE>
    </IF>

   </THEN>
   <ELSE>

    <H3>Sorry, can't find your credentials.</H3>

   </ELSE>
</IF>
```

# Creating User Account Creation Forms

This section provides simple code samples that illustrate how to code forms that prompt a visitor to register (obtain a user account) and then create a user account for that visitor.

This example presents code from the following elements:

- `PromptForNew Account`, an XML element that displays a form that requests the visitor to enter the user name and password that he or she would like to use
- `CreateAccount`, a JSP element that creates the new account

## PromptForNewAccount

The `PromptForNewAccount` element displays a form that prompts the visitor to enter a user name and password and to re-enter the password to confirm it.

Here’s the code that creates the form:

```
<div align="center">
<h3>Create a New Account</h3>
<FORM ACTION="ContentServer" METHOD="post">
```

```
<input type="hidden" name="pagename" value="CSGuide/Security/
CreateAccount"/>

<table cellpadding="5" cellspacing="5">

<tr>
<th align="right">Pick a username</th>
<td><input type="text" name="username" size="16"/></td>
</tr>

<tr>
<th align="right">Pick a password</th>
<td><input type="password" name="password" size="16"/></td>
</tr>

<tr>
<th align="right">Confirm your new password</th>
<td><input type="password" name="confirm_password" size="16"/></
td>
</tr>

<tr>
<td> </td>
<td><input type="submit" name="doit" value="Create Account"/></td>
</tr>
</table>

</FORM>
</div>
```

The visitor fills in the form and clicks the **Submit** button. The information gathered in the form and the page name of the `CreateAccount` page (see the first `input type` statement, above) is sent to the browser.

The browser sends the page name to Content Server, Content Server looks it up in the `SiteCatalog` table and then invokes that page entry's root element.

## Root Element for the CreateAccount Page

There can only be one root element for a Content Server page (that is, an entry in the `SiteCatalog` table). The root element for the `CreateAccount page` is the `CreateAccount.jsp` element.

Only someone with `SiteGod` or `ContentEditor` ACLs can create a new user account. Because of this restriction, the `CreateAccount` element does the following:

- Logs the visitor in as a privileged user, without the knowledge of the visitor.
- Creates the account.
- Assigns the new user the appropriate ACLs (every user must belong to at least one ACL)

Here's the code that creates the new user account:

```
<SETVAR NAME="errno" VALUE="0"/>
```

```
<!-- switch temporarily to a privileged user -->
<!-- The username and password for the privileged user should be
encrypted in a property file. You should obtain them from the
property file, decrypt them, then pass them it. For this example,
they are hard-coded. -->

<USER.SU USERNAME="jumpstart" PASSWORD="jumpstart">

<USERISMEMBER GROUP="UserEditor"/>
<IF COND="Variables.errno!=1">
   <THEN>
         <h3>An error has occurred creating the account (no
   UserEditor
         privs).  Contact the webmaster</h3>
   </THEN>
   <ELSE>
    <IF COND="Variables.password!=Variables.confirm_password">
         <THEN>
         <h3>Your passwords do not match. Click the Back button
   and
         try again.</h3>
         </THEN>
         <ELSE>

   <!-- Get the parameters from the property file -->

            <ics.getproperty name="username" file="dir.ini"
            output="unameattr"/>
            <ics.getproperty name="password" file="dir.ini"
            output="passattr"/>

   <!-- create the user's name in the right format for the dir
   tags -->

            <ics.getproperty name="peopleparent" file="dir.ini"
            output="namebase"/>
            <name.makechild context="Variables.namebase"
   output="iname">
               <name.argument name="Variables.unameattr"
               value="Variables.username"/>
            </name.makechild>

   <!-- create the user -->

            <dir.create name="Variables.iname">
            <dir.argument name="Variables.unameattr"
            value="Variables.username"/>
            <dir.argument name="Variables.passattr"
            value="Variables.password"/>

   <!-- additional parameters can be added here but for the
   example we won't -->
```

```
<!-- In particular, if you are using LDAP, you will have to
spin through and set the values of the properties in the
property requiredPeopleAttrs in dir.ini. -->

        </dir.create>

        <IF COND="Variables.errno=0">
        <THEN>

<!-- give the new user an acl and format it correctly for
dir.addgroupmember -->

            <ics.getproperty name="groupparent" file="dir.ini"
            output="groupparent"/>
            <ics.getproperty name="cn" file="dir.ini"
output="cn"/>
            <name.makechild context="Variables.groupparent"
            output="groupid">
               <name.argument name="Variables.cn"
value="Browser"/>
            </name.makechild>

<!-- add the acl -->

            <dir.addgroupmember name="Variables.groupid"
            member="Variables.iname"/>

            <IF COND="Variables.errno=0">
               <THEN>
                <h3>Success!</h3>
               </THEN>
               <ELSE>
                  <h3>User created but error adding user to
group.
                  Contact the webmaster</h3>
               </ELSE>
            </IF>

         </THEN>
         <ELSE>
            <h3>Error creating user!  Contact the webmaster.</
h3>
         </ELSE>
      </IF> <!-- create success check -->

      </ELSE>
</IF> <!-- passwords match -->

</ELSE>
</IF>

</USER.SU>
```

# Visitor Access in the Burlington Financial Sample Site

The Burlington Financial sample site includes a membership component that uses elements to sign up new members and log in existing members.

These visitor registration elements are not robust enough for use on a real-world web site, but can give you a starting point for your own designs. For example, Burlington Financial has sample visitor account screens, allowing visitors to register and set their own preferences, but does not use this information to restrict visitor access to certain web pages, or to make recommendations based on a member's profile.

## Membership Table

Burlington Financial uses a table named `bfmembers` to implement the membership component. (This table is created for the sample site when it is installed—none of the CS modules or products use this table.) Although the membership elements add a row to the `bfmembers` database table for each new registered member's profile information, they do not add a row to the `SystemUsers` table.

## Users and Passwords

There is one generic user, BFUser, for all Burlington Financial members. The name and password are the same (`BFUser/BFUser`) and should not be changed. The member login code in Burlington Financial sets a session variable for the visitor, which is then used to identify that visitor.

Because Burlington Financial is a sample site, members' passwords are stored in the `bfmembers` table as plain text. A real web site would store passwords in encrypted format. Burlington also grants Visitor, BFMember, and Browser ACL privileges to entries added to the `bfmembers` table.

## Member Accounts

There are currently no elements for managing the Burlington Financial accounts. If you want to try editing or deleting members' accounts, use Content Server Explorer to modify the `bfmembers` table.

## Membership Processing Elements

There are several elements that handle processing requests for Burlington Financial members. If you have installed the Burlington Financial sample site, you can use Content Server Explorer to open and examine them. All but one is located here:

```
ElementCatalog/BurlingtonFinancial/Util/Account
```

The `AccountAccessScript` element is located here:

```
ElementCatalog/BurlingtonFinancial/Util
```

### AccountAccess.xml

This is a page template that calls pagelet elements for the header, footer, navigation menu, and the account content.

## AccountAccessScript.xml

This file contains three JavaScript routines (`checkSignupForm`, `checkProfileForm`, and `checkLoginForm`) that perform basic error checking on the HTML account forms. This is called from `Login.xml`, `Profile.xml`, and `SignUp.xml` elements.

## Benefits.xml

This page calls the `Block.xml` article template to render an article of text about the Burlington Financial site. On a real web site, the article would contain benefits information.

## Login.xml

This page displays the login screen for registered members and calls `LoginPost.xml` to handle the login form input. It also calls `Benefits.xml`, and `SignUp.xml` for non-members.

## LoginPost.xml

This pagelet element calls `ProcessLogin.xml` to display a login message.

## Profile.xml

This page displays an editable profile form if the visitor is registered, or else calls `SignUp.xml` if the visitor is not registered.

## ProcessLogin.xml

This pagelet element displays an appropriate login message, depending on whether the visitor who submitted the form is a registered member.

## SignUp.xml

This page displays the sign-up screen for non-registered visitors and calls `the catalogmanager` to add a row to the `bfmembers` table for a new user, or to update the `bfmembers` table for an existing user.

Chapter 32

# The HelloAssetWorld Sample Site

The HelloAssetWorld sample site is a sample web site built using Content Server and CS-Direct. It is meant to provide a simple entry point into the process of building a web site with CS-Direct. This chapter focuses on the steps that a developer would take in creating this simple web site; further information on HelloAssetWorld's configuration and users are in the *Content Server Administrator's Guide*.

This chapter contains the following sections:


- Overview
- Modified Asset Types
- HelloAssetWorld Templates
- The HelloQuery Asset

# Overview

The HelloAssetWorld site has a simple design; it is composed of one page, as shown in the following screen capture:

The stories that appear on this web page change depending upon the article that you choose to view, but the layout of the page remains the same.

To view the HelloAssetWorld sample site yourself, enter the following URL into your web browser:

`http://server_name/servlet/ContentServer?pagename=HelloAssetWorld/Page/HelloPageTemplate`

## HelloAssetWorld Templates

The HelloAsset World web page is composed of three templates:

- The HelloArticle template, which displays the article that you select and that article's associated image.
- The HelloCollection template, which displays hyperlinks to a collection of articles that you can view.
- The HelloPage template, which is the containing page. It displays the HelloAssetWorld banner graphic and calls the HelloArticle and Hello Collection templates.

## HelloAssetWorld Asset Types

The asset types used in the HelloAssetWorld site are modified from asset types used in the Burlington Financial sample site, described in Chapter 33, "The Burlington Financial Sample Site." A list of the asset types used in HelloAssetWorld follows:

- The Page asset type, which performs several functions:
  - It allows uses to create a site hierarchy by **placing** the page. Placing a page gives it an entry in the `SitePlanTree` table and allows it to be viewed under the Placed Pages node of the Site Plan Tree.
  - It allows you to associate assets of various types with it. For example, you can associate Collection assets and Article assets with a Page asset.

    The instance of the Page asset type used in the HelloAssetWorld site, HelloPage, has a collection called HelloCollectionHello associated with it. The Page asset type is a core asset type which is provided with CS-Direct, and has not been modified.
- The HelloArticle asset type, which contains an article. HelloArticle assets can have HelloImage assets associated with them. The HelloArticle asset type has been modified from the Article asset type that is provided with Burlington Financial.
- The HelloImage asset type, which contains an image. The HelloImage asset type has been modified from the ImageFile asset type that is provided with Burlington Financial.
- The Query asset type, which queries the database and returns the HelloArticle assets that display on the web site. This is a core asset type, which is provided with CS-Direct and has not been modified.
- The Collection asset type, which orders the results that the query asset returns. This is a core asset type, which is provided with CS-Direct and has not been modified.
- The Template asset type, which renders the various asset types. This is a core asset type, which is provided with CS-Direct and has not been modified.

# Modified Asset Types

Most of the asset types used in the HelloAssetWorld sample site are core asset types, and hence cannot be modified. The HelloArticle and HelloImage asset types, however, are simplified versions of the Article and ImageFile asset types that are provided with CS-Direct. Each asset type has a new asset descriptor file that is based on the asset descriptor files for the Article and ImageFile asset types. The simplified asset descriptor files are shown in the following sections.

## The HelloArticle Asset Type

The `ASSET` tag, shown in the following line, is the standard opening for all asset descriptor files. Among other things, it names the new asset type and specifies the asset's defdir, the default directory where uploaded items are stored.

```
<ASSET NAME="HelloArticle" DESCRIPTION="HelloArticle"
MARKERIMAGE="/Xcelerate/data/help16.gif" PROCESSOR="4.0"
DEFDIR="c:\FutureTense\Storage\HelloArticle">
```

The next lines create a text field for the article's headline.

```
<PROPERTIES>
<PROPERTY NAME="Headline" DESCRIPTION="Headline">
<STORAGE TYPE="VARCHAR" LENGTH="255" />
<INPUTFORM TYPE="TEXT" WIDTH="48" MAXLENGTH="255"
REQUIRED="YES" />
<SEARCHFORM DESCRIPTION="Headline contains" TYPE="TEXT"
WIDTH="48" MAXLENGTH="255" />
</PROPERTY>
```

The next lines create a text field for the article's byline.

```
<PROPERTY NAME="Byline" DESCRIPTION="Byline">
<STORAGE TYPE="VARCHAR" LENGTH="100" />
<INPUTFORM TYPE="TEXT" WIDTH="48" MAXLENGTH="100"
REQUIRED="YES" />
<SEARCHFORM DESCRIPTION="Byline contains" TYPE="TEXT"
WIDTH="48" MAXLENGTH="100" />
</PROPERTY>
```

The following lines create an upload field where content editors and authors can type in the content of an article's body. This content will be stored in the defdir specified in the `ASSET` tag.

```
<PROPERTY NAME="urlBody" DESCRIPTION="Body">
<STORAGE TYPE="VARCHAR" LENGTH="2000" />
<INPUTFORM TYPE="TEXTAREA" COLS="300" ROWS="300" REQUIRED="YES"
/>
</PROPERTY>
</PROPERTIES>
</ASSET>
```

## The HelloImage Asset Type

The `ASSET` tag, shown in the first line below, is the standard opening for all asset descriptor files. Among other things, it names the new asset type and specifies the asset's defdir, the default directory where uploaded items are stored.

```
<ASSET NAME="HelloImage" DESCRIPTION="HelloImage"
MARKERTEXT="*" PROCESSOR="4.0"
DEFDIR="c:\FutureTense\Storage\HelloImage">
```

Then, the next lines create an upload field for the image file.

```
<PROPERTIES>
<PROPERTY NAME="urlfile" DESCRIPTION="Image File">
<STORAGE TYPE="VARCHAR" LENGTH="255"/>
<INPUTFORM TYPE="UPLOAD" WIDTH="36" REQUIRED="NO"
LINKTEXT="HelloImage"/>
</PROPERTY>
```

The following lines create a drop-down select and specify how the search field for mimetypes will appear on the "Advanced Search" form. The SQL statement supplied as a value for the `SQL` parameter for the `INPUTFORM` tag queries the database to supply mimetypes for the text of the dropdown.

```
<PROPERTY NAME="mimetype" DESCRIPTION="Mimetype">
```

```
<STORAGE TYPE="VARCHAR" LENGTH="36"/>
<INPUTFORM TYPE="SELECT" SOURCETYPE="TABLE"
TABLENAME="mimetype" OPTIONDESCKEY="description"
OPTIONVALUEKEY="mimetype" SQL="SELECT mimetype, description
FROM mimetype WHERE keyword = 'image' AND isdefault = 'y'"
INSTRUCTION="Add more options to mimetype table with
isdefault=y and keyword=image"/>
```

The next line specifies how the mimetype field will appear on the “Advanced Search” form. As shown above, the SQL supplied here queries the database for mimetypes to fill the dropdown select with.

```
<SEARCHFORM DESCRIPTION="Mimetype" TYPE="SELECT"
SOURCETYPE="TABLE" TABLENAME="mimetype"
OPTIONDESCKEY="description" OPTIONVALUEKEY="mimetype"
SQL="SELECT mimetype, description FROM mimetype WHERE keyword =
'image' AND isdefault = 'y'"/>
</PROPERTY>
```

The following lines create a text field that allows users of the management system to input alternate text for the image. The `SEARCHFORM` tag specifies how the `Alt Text contains` field will appear on the “Advanced Search” form.

```
<PROPERTY NAME="alttext" DESCRIPTION="Alt Text">
<STORAGE TYPE="VARCHAR" LENGTH="255"/>
<INPUTFORM TYPE="TEXT" WIDTH="48" MAXLENGTH="255"
REQUIRED="NO"/>
<SEARCHFORM DESCRIPTION="Alt Text contains" TYPE="TEXT"
WIDTH="48" MAXLENGTH="255"/>
</PROPERTY>

</PROPERTIES>
</ASSET>
```

# HelloAssetWorld Templates

The HelloAssetWorld sample site uses three Template assets to render the assets that were described previously. The following sections describe these Template assets.

## The HelloArticle Template

The HelloArticleTemplate renders HelloArticle assets. The template uses the following variables:

| Variable | Value | Source |
|---|---|---|
| `tid` | The current template's ID. | The `tid` variable is set in the `resargs1` field of the SiteCatalog table. The value is set automatically when the template is created. |
| `c` | The type of content that the template displays. | The `c` variable is set in the `resargs1` field of the SiteCatalog table. The value is set using the `Asset Type` field on the "New Template" form. |
| `cid` | The ID of the asset to load. | The `cid` variable is passed in by the HelloPage template. |
| `picture:oid` | The object ID of a HelloImage asset that is associated with the HelloArticle asset. | The `picture:oid` variable is obtained by loading the current HelloArticle asset and using the `ASSET.CHILDREN` tag to find information on associated HelloImage assets. |
| `picture:alttext` | The alternate text for the `associated HelloImage` asset. | The `picture:alttext` variable is obtained by loading the current HelloArticle asset and using the `ASSET.CHILDREN` tag to find information on associated HelloImage assets. |
| `picture:mimetype` | The mimetype of the associated `HelloImage` asset. | The `picture:mimetype` variable is obtained by loading the current HelloArticle asset and using the `ASSET.CHILDREN` tag to find information on associated HelloImage assets. |
| `asset:headline` | The value in the `Headline` field of this HelloArticle asset. | The `asset:headline` variable is obtained by scattering the information in the `HelloArticle` asset. |
| `asset:byline` | The value in the `Byline` field of this HelloArticle asset. | The `asset:headline` variable is obtained by scattering the information in the `HelloArticle` asset. |

| Variable | Value | Source |
|---|---|---|
| artID | The ID of the article to display in the HelloArticle template. | On the first page view, this is set in the resdetails field of the ElementCatalog entry. On subsequent viewings, this is passed in by the HelloCollection template. |

The following lines are the standard beginning for an article template. They appear when you click the XML or JSP buttons on the “New Template” form in the Content Server user interface.

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!--  HelloArticle/HelloArticleTemplate
-
- INPUT
-
- OUTPUT
-
-->
```

The RENDER.LOGDEP tag marks the template as a cache dependency item. This means that when the template is modified, any outdated copies of the template will be removed from the Content Server and Satellite Server caches and replaced with current versions automatically.

```
<IF COND="IsVariable.tid=true">
<THEN>
      <RENDER.LOGDEP   cid="Variables.tid" c="Template"/>
</THEN>
</IF>

<table border="0" cellspacing="2" cellpadding="2">

<tr>
<td>
```

The following ASSET.LOAD tag loads a HelloArticle asset using the asset’s ID, which is stored in Variables.cid. This value is passed in to the HelloArticle template by the HelloPage template.

```
<!-- asset load will mark the asset as an 'exact' dependent of
the pagelet being rendered -->

<ASSET.LOAD NAME="helloArticleAsset" TYPE="Variables.c"
OBJECTID="Variables.cid"/>
```

The next line uses the ASSET.CHILDREN tag to load any HelloImage assets that are associated with the article. ASSET.CHILDREN creates a list which contains the information necessary to display the HelloImage asset.

```
<ASSET.CHILDREN NAME="helloArticleAsset" LIST="picture"
TYPE="HelloImage"/>
```

These lines check to see if there is a HelloImage asset associated with the current article. If there is no associated HelloImage asset, the template only displays the text of the article.

```
<!--Check to see if the list (which contains information to
display an image) exists--if the list doesn't exist, display
the article text only. -->
<IF COND="IsList.picture=true">
  <THEN>
  <!--Log the image as a dependency.-->
  <RENDER.LOGDEP   cid="picture.oid" c="HelloImage"/>
```

These lines use the `RENDER.SATELLITEBLOB` tag to display the associated image.

```
  <!--Display the image.-->
  <RENDER.SATELLITEBLOB BLOBTABLE="HelloImage" BLOBKEY="id"
BLOBCOL="urlfile" BLOBWHERE="picture.oid"
BLOBHEADER="picture.mimetype" SERVICE="IMG SRC"
ARGS_ALT="picture.alttext" ARGS_ALIGN="left"/>
  </THEN>
</IF>
</td>
</tr>

<tr>
<td>
```

Then this `ASSET.SCATTER` tag gets all of the HelloArticle asset's primary fields.

```
<!-- get all the primary table fields of the asset -->
<ASSET.SCATTER NAME="helloArticleAsset" PREFIX="asset"/>
```

The following `CSVAR` tag displays the contents of the asset's fields.

```
<!-- display the headline-->

<h3><CSVAR NAME="Variables.asset:headline"/></h3>
</td>
</tr>

<tr>
<td>
<CSVAR NAME="Variables.asset:byline"/>
</td>
</tr>

<tr>
<td>
```

Because the Body field may contain an embedded link, it must be retrieved using the `ics.getvar` tag and displayed using the `RENDER.STREAM` tag, shown in the following lines:

```
<!-- display the body-->
<ics.getvar name="asset:urlbody" encoding="default"
output="bodyvar"/>
<RENDER.STREAM VARIABLE="bodyvar" /><br/>
</td></tr>
```

```
</table>

</FTCS>
```

## The HelloCollection Template

The collection template displays the HelloCollectionHello collection. It uses the following variables:

| Variable | Value | Source |
|---|---|---|
| `tid` | The current template's ID. | The `tid` variable is set in the `resargs1` field of the SiteCatalog table. The value is set automatically when the template is created. |
| `c` | The type of content that the template displays. | The `c` variable is set in the `resargs1` field of the SiteCatalog table. The value is set using the `Asset Type` field on the "New Template" form. |
| `cid` | The ID of the asset to load. | The `cid` variable is passed in by the HelloPage template. |
| `tid` | The current template's ID. | The `tid` variable is set in the `resargs1` field of the SiteCatalog table. The value is set automatically when the template is created. |
| `p` | The ID of the page that generates the hyperlink—in this case, the current template. | |
| `ApprovedArticles:id` | The ID of the current HelloArticle in the collection. | |
| `ApprovedArticles:name` | The name of the current HelloArticle in the collection. | |
| `artID` | On line 23 this variable contains the ID of the HelloArticle being displayed by the HelloArticleTemplate. On line 33, the value is the ID of the current article in the collection. | The variable used on line 23 is passed in by the HelloPage template. The variable set on line 33 gets its value from the ID of the current HelloArticle in the collection. It is then passed to the HelloPage and HelloArticle templates. |
| `pageID` | The ID of the HelloPage asset. | |

| Variable | Value | Source |
|---|---|---|
| referURL | The URL generated by the RENDER.GETPAGEURL tag. | |

```
<IF COND="IsVariable.tid=true">
<THEN>
      <RENDER.LOGDEP   cid="Variables.tid" c="Template"/>
</THEN>
</IF>
```

The following ASSET.LOAD tag loads the HelloCollectionHello asset, based upon the value of its ID, contained in Variables.cid. Variables.cid is passed in by the HelloPage template. ASSET.LOAD also names the asset HelloCollection. This does not change the name of the asset in the database; rather it sets the name which the rest of the code in the template uses to refer to the collection asset.

```
<ASSET.LOAD NAME="HelloCollection" TYPE="Collection"
OBJECTID="Variables.cid"/>
<ASSET.SCATTER NAME="HelloCollection" PREFIX="asset"/>
```

Then the following ASSET.CHILDREN tag loads a list containing the HelloArticles that compose the collection.

```
<ASSET.CHILDREN NAME="HelloCollection" LIST="theArticles"
OBJECTTYPE="HelloArticle"/>
```

The next line uses the RENDER.FILTER tag to filter out articles that are not approved for Export to Disk Publishing. This allows the template to be used for both Mirror Publishing and Export to Disk Publishing.

```
<!--Filter out assets which aren't approved for export to disk
publishing.-->
<RENDER.FILTER LIST="theArticles"
LISTVARNAME="ApprovedArticles" LISTIDCOL="oid"/>
```

In the following lines of code, the LOOP tag loops through the list of approved article and the RENDER.LOGDEP tag logs each item in the list as a cache dependency.

```
3   <LOOP LIST="ApprovedArticles">
4   <RENDER.LOGDEP cid="ApprovedArticles.id" c="Article"/>
```

The following lines use an IF tag to check whether the current article in the list is the article being displayed by the HelloArticle template. The ID of the article being displayed by the HelloArticle template is contained in Variables.artID. This variable is passed in by the HelloPage template. If the article IDs are the same, the name of the article is displayed in bold text and is not a hyperlink.

```
<IF COND="Variables.artID=ApprovedArticles.id">
<THEN>
<B><CSVAR NAME="ApprovedArticles.name"/></B><P/>
</THEN>
```

If the current article in the list is not the article being displayed by the HelloArticle template, then a URL is generated to create a hyperlink to that article. The URL is created

by the RENDER.GETPAGEURL tag in the following lines. RENDER.GETPAGEURL appends the artID variable to the URL that it creates. This variable contains the ID of the article to display.

```
 <ELSE>
 <RENDER.GETPAGEURL PAGENAME="HelloAssetWorld/Page/
HelloPageTemplate"
cid="Variables.pageID"
c="Page"
p="Variables.p"
OUTSTR="referURL"
ARGS_artID="ApprovedArticles.id"/>
```

These lines display the hyperlink. The REPLACEALL argument evaluates Variables.referURL, which contains the URL for the hyperlink. The CSVAR tag, used on line 38, displays the name of the article that the hyperlink links to.

```
<A HREF="Variables.referURL" REPLACEALL="Variables.referURL">
<CSVAR NAME="ApprovedArticles.name"/>
</A><P/>
</ELSE>
</IF>

</LOOP>

</FTCS>
```

## The HelloPage Template

The HelloPage template acts as a containing page. It renders the HelloPage asset, displays a header graphic, and calls the HelloCollection and HelloArticle templates, creating the finished page layout.

The HelloPage template uses the following variables:

| Variable | Value | Source |
|---|---|---|
| `tid` | The current template's ID. | The `tid` variable is set in the `resargs1` field of the SiteCatalog table. The value is set automatically when the template is created. |
| c | The type of content that the template displays. | The `c` variable is set in the `resargs1` field of the SiteCatalog table. The value is set using the `Asset Type` field on the "New Template" form. |
| cid | The ID of the asset to load. | The `cid` variable is set in the `resargs1` field of the template's SiteCatalog entry. |
| `topImg:ID` | The ID of the `TopImage` ImageFile asset. | The `topImg:ID` variable is obtained by scattering the information in the `TopImage` image file asset. |
| `topImg:alttext` | The alternate text for the `TopImage` asset. | The `topImg:alttext` variable is obtained by scattering the information in the `TopImage` image file asset. |
| `topImg:mimetype` | The mimetype of the `TopImage` asset | The `topImg:mimetype` variable is obtained by scattering the information in the `TopImage` image file asset. |
| `asset:ID` | The ID of the Page asset that this template renders. | The `asset:ID` variable is obtained by scattering the information in the `HelloPage` asset. |
| theCollection.oid | The ID of the HelloCollectionHello collection, to be passed to the HelloCollection template for display. | The collection, and hence its ID, are associated with the Page asset. |
| `artID` | The ID of the article to display in the HelloArticle template. | On the first page view, this is set in the `resdetails` field of the ElementCatalog entry. On subsequent viewings, this is passed in by the HelloCollection template. |

The code for the HelloPage template follows, along with a description of what it does:

```
<IF COND="IsVariable.tid=true">
<THEN>
      <RENDER.LOGDEP cid="Variables.tid" c="Template"/>
</THEN>
</IF>

<!--Table for formatting-->
<table border="1" cellpadding="5" cellspacing="5">

<tr>
<td colspan="2">
```

These lines load a HelloImage asset and display that asset using the `RENDER.SATELLITEBLOB` tag.

```
<!--Embedded Image Asset-->

<!-- The following 2 lines line load the image file and scatter
the information in its fields. -->
<ASSET.LOAD NAME="TopImage" TYPE="HelloImage"
OBJECTID="1024605735822"/>
<ASSET.SCATTER NAME="TopImage" PREFIX="topImg"/>

<!-- This line creates a URL to display the image file.-->
<RENDER.SATELLITEBLOB BLOBTABLE="HelloImage" BLOBKEY="id"
BLOBCOL="urlfile" BLOBWHERE="Variables.topImg:id"
BLOBHEADER="Variables.topImg:mimetype" SERVICE="IMG SRC"
ARGS_alt="Variables.topImg:alttext" ARGS_WIDTH="600"
ARGS_HEIGHT="90"/>
</td>
</tr>


<tr>
<td>
```

Next, the code loads the HelloPage asset based on the value of `Variables.cid`, which is set in the `resargs1` field of the template's SiteCatalog entry.

```
<!-- This loads the HelloPage asset and names it HelloPage. -->
<ASSET.LOAD NAME="HelloPage" TYPE="Variables.c"
OBJECTID="Variables.cid"/>

<!-- This scatters the fields of the HelloPage asset for use
later in the element. -->
<ASSET.SCATTER NAME="HelloPage" PREFIX="asset"/>


<!-- This finds the collection asset associated with the
HelloPage asset and puts the information the collection into a
list. -->
<ASSET.CHILDREN NAME="HelloPage" LIST="theCollection" />
```

```
<RENDER.LOGDEP C="Collection" CID="theCollection.oid"/>
<!-- This checks to see whether ASSET.CHILDREN really generated
a list. -->
<IF COND = "IsList.theCollection=true">

   <THEN>
         <!-- This displays the HelloCollectionTemplate
template and passes the template the ID of the collection to
display and the ID of the current page. -->
         <RENDER.SATELLITEPAGE PAGENAME="HelloAssetWorld/
Collection/HelloCollectionTemplate"
ARGS_cid="theCollection.oid" ARGS_p="Variables.asset:id"
ARGS_artID="Variables.artID"/>
   </THEN>

</IF>
</td>

<td>
<!-- This displays the HelloArticleTemplate template and passes
the template the ID of the article to display and the ID of the
current page. The artID variable is passed in by the URL.-->
<RENDER.SATELLITEPAGE PAGENAME="HelloAssetWorld/HelloArticle/
HelloArticleTemplate" ARGS_cid="Variables.artID"
ARGS_p="Variables.asset:ID"/>
</td>
</tr>
</table>

</FTCS>
```

# The HelloQuery Asset

The HelloAssetWorld site uses a query asset named HelloQuery to retrieve HelloArticles from the database. A content provider then creates and builds a collection, ranking the items that the HelloQuery asset returns in the order that they will be displayed.

Chapter 33

# The Burlington Financial Sample Site

The Burlington Financial sample site demonstrates site design best practices using CS-Direct assets.

Content management can be highly abstract, especially when described in terms of assets, queries and templates. Burlington Financial helps you to understand what the end result of your application can be—a live, functioning web site. Developers and content managers have immediate and easy access to the sample site's code and queries.

This chapter contains the following sections:


- Overview
- Navigation Features
- Best Practices

# Overview

Burlington Financial is a fictitious financial news site. The site emphasizes navigation between sections, the site's hierarchy, how the site works with CS-Direct and CS-Direct Advantage asset types, and a real-world look-and-feel. Burlington Financial also has about five hundred articles and over a hundred images, or enough real-world content to populate several sections.

Burlington Financial is a fully functional sample site with the following features:

- Includes search, member login, printer-friendly articles, e-mail to a friend, topic directory, and stylesheets
- Demonstrates a hierarchy of web site sections
- Supports component caching and Satellite Server
- Demonstrates the use of assets created with AssetMaker
- Demonstrates the use of CS-Direct Advantage Flex Assets and Engage Segment Assets
- Includes a meaningful amount of content
- Approximates a real-world site that developers can learn from

The Burlington Financial home page is shown in the following figure:

Burlington Financial takes advantage of cacheable pagelets. These individual pagelets can be cached and managed independently, giving developers greater performance and flexibility on the site.

FatWire encourages you to design your site using cacheable pagelets. For more information, see Chapter 5, “Page Design and Caching.”

# Navigation Features

The following three elements are used to display the primary navigation bars in Burlington Financial (which you can look at using Content Server Explorer):

### BurlingtonFinancial/Site/TopSiteBar.xml

This element draws the hyperlinks to the home page and its top-level children at the top of the page, just under the Burlington Financial logo.

It intentionally displays only the top-level children of the Home page, so that the row of hyperlinks does not wrap, breaking the design of the page. The Home page appears first and it is at the same level as its children.

### BurlingtonFinancial/Site/LeftSideSiteBar.xml

This element draws a more detailed map of the major sections of the web site and looks at the child and grandchild pages of the Home page. This element could be modified to go more than two levels deep, although the graphic design of the site limited the space available.

Notice that two other major pages are listed here that were not listed in the `TopSiteBar`—the Wire Feed and Columnists pages are independent of the Home page and its children, and are displayed separately.

### BurlingtonFinancial/Site/BottomNavFooter.xml

This element draws the hyperlinks at the bottom of every page. Like `LeftSideSiteBar`, the `BottomNavFooter` element also includes links to pages that are not children of the Home page.

**Note**

If you add another Page asset to the site, and it is not a descendant of the Home Page asset, then it will not automatically appear in any of the navigation bars used in Burlington Financial.

Although these navigation bars are computed dynamically, in a real-world web site they would probably not change very often. For maximum performance, you could simply replace the dynamic code in the element with a static list of hard-coded links to the top-level pages. Later on, if you do need to change the site and add a new top-level section, you need only modify a few elements.

If you use dynamic navigation bar elements, you should set a long cache time-out for the navigation bar pagelets.

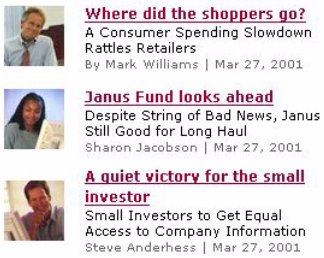
## Breadcrumbs

This common feature in web sites is a tiny map of the path to a particular item. In Burlington Financial, it is a conceptual path rather than the actual history of pages visited, and is located just under the top navigation bar:

Because CS-Direct allows assets to be assigned to multiple parents at the same time, the same article can appear as a member of a collection on the Home page and on the News page at the same time. There is no way to identify the true parent of the article, so Burlington Financial passes the `id` of the parent Page anytime it draws a hyperlink to a child. That way, when the child asset draws itself, it already has the ID of the desired parent. This value is passed in the variable `p`.

Since our templates generate different HTML based on different values for `p`, those versions should be cached independently. So the variable `p` must be part of the page criteria variables listed in the page entry in the `SiteCatalog` for that template. Because this is such a common technique, CS-Direct automatically includes `p` in the list of page criteria variables for a Template asset's `SiteCatalog` page entry.

Sometimes you may want to override `p` and use a different parent asset. For example, the Burlington Financial Home page has links to articles by the following columnists:




However, these articles don't belong conceptually to the Home page—rather, they belong to the Columnists page. So you can load and pass the ID of the Columnists Page asset as

the parent for those hyperlinks. This way, when a visitor clicks on them, the breadcrumb identifies Columnists as the parent. This behavior is consistent with clicking on the Columnists link in the navigation bar, and then clicking on one of the articles there:

Columnists ▸ Where did the shoppers go?

In other cases, it isn't immediately clear which asset should be the parent. For example, Burlington Financial treats the articles in the “From the Wires” box as belonging to the current page. Stories listed on the main “Wire Feed” section page belong to the “Wire Feed” section, and show Wire Feed as their parent.

# Best Practices

The Burlington Financial sample site demonstrates Content Server best practices for several other important features found in real-world web sites.

## Searching

The database search code in Burlington Financial is very similar to the search code in the CS-Direct application itself. It works with dynamic delivery, but not with exported static HTML. In this case, you’ll need a different search mechanism for indexing the static HTML files.

The `BurlingtonFinancial/Util/SearchPost` element uses SQL searching against the Article table, or it can use the search engine index if it is installed and enabled for the Article asset type. SQL searching is case sensitive. Using a search engine would allow more sophisticated search capabilities, such as case-insensitive searching, word variants and word stemming.

## Keywords

The Article asset type contains a field called **keyword** which lets editors associate specific terms with an Article for improved searching of the Article asset type. Burlington Financial Article assets have one or more keywords separated by commas, for example, “Energy, Shell Oil, OPEC.” Burlington Financial uses keywords to display lists of Hot Topics.

## Hot Topics

Burlington Financial Hot Topics demonstrate one use of query assets.

On the left side of most pages, there is a list of Hot Topics for a particular section of the site. Hot topics are listed according to which section the visitor is viewing, as determined by the Page assets.

In the following example, the Hot Topics in the News section are Human Genome, History, Sanctions, Energy and California:

The element `BurlingtonFinancial/Common/LeftNavColumn` includes the pagelet `BurlingtonFinancial/Query/ShowHotTopics`. This element receives the ID of a page asset, passed through the variable `p`. If it cannot find a value for `p`, it defaults to using the Home page. It loads that page asset and looks for the top stories collection associated with the page and loads it. The element then loops through each of the articles in the collection and builds a list of keywords, pulled from the keyword field of the article (multiple keywords for an article must be delimited by commas). After it has made a list of the keywords, the element loops over that list, listing each keyword as a hyperlink to a page that runs a query for that keyword, by rendering the query asset named HotTopics.

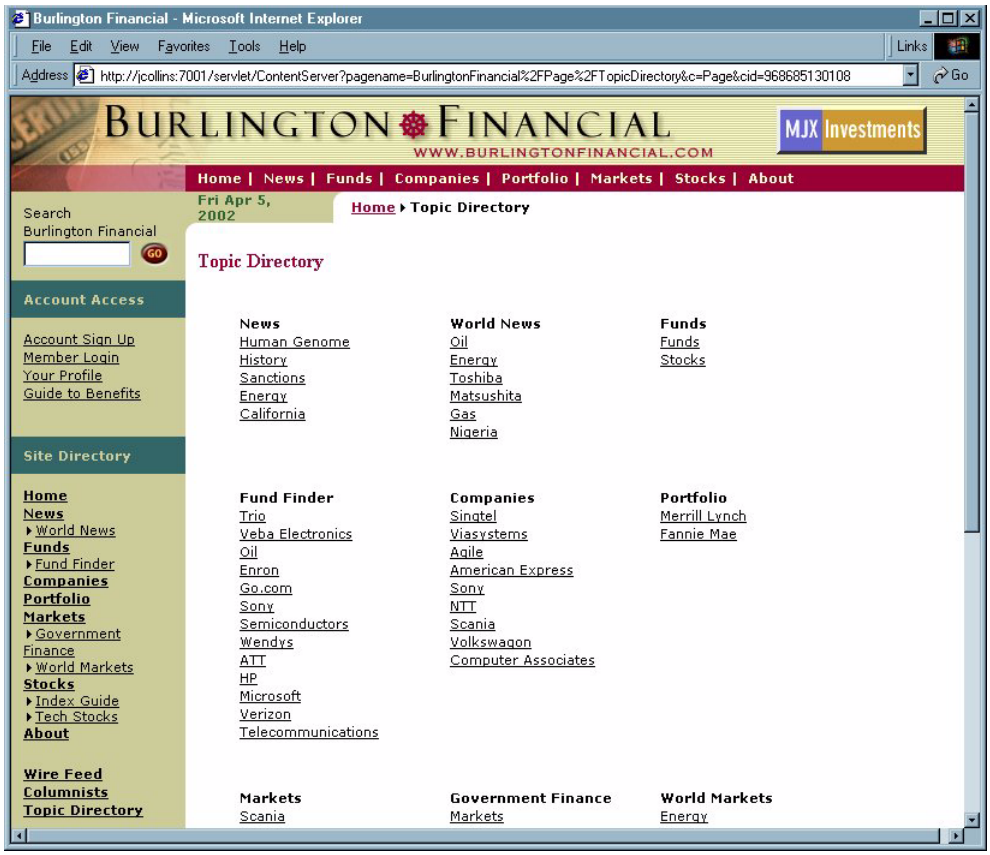
When visitors click a link, they are taken to a page that renders the query asset HotTopics, using the HotTopicFront template. The HotTopics query does a straight SQL match against articles that contain the selected keyword. The keyword search is not constrained in any way—it searches all articles in the Burlington Financial site, not just those in a particular section or category.

Each article returned by the HotTopics query inherits the parent ID of the page asset where the visitor first started looking at the keywords. Clicking on the Shell Oil story from the list of Energy stories under the News page causes the story to be displayed with News as its parent page. Clicking on the same Shell Oil story from the list of Energy stories under the World News page causes the story to be displayed with World News as its parent.

This design is not a problem in the dynamic live site; however it does cause duplicate files to be exported to a static delivery site.

## Topic Directory

At the bottom of the left navigation column, and also in the navigation links at the bottom of each page, there is a hyperlink to the Topic Directory:




This page consists of the Hot Topics pagelet for every section. Since the pagelet that is used here is also used in the left navigation column, it is displayed in the browser very quickly. Each topic inherits its parent page and passes it to the list of articles the query returns.

## Related Stories

The query asset used for an article's Related Stories list is similar to the previously described HotTopics query, but instead of getting the keywords from another page, it gets the keyword from the article that the query is associated to. The Article template “Full” includes the Related Stories pagelet, and passes the Article's first keyword to the query asset. When the Related Stories query is executed, it looks for other articles with the same keyword.

For example, from the Home page, click on the Hot Topic “Microsoft” and choose the story “Microsoft launches worldwide campaign against counterfeit software.” This article (cid=984156689788) has the world “Microsoft” in its keyword field. It also has the Related Keyword query associated with it. When this Query is executed during rendering, the SQL looks up five other Articles with the term “Microsoft” in their keyword field. The query is designed to exclude the article that it is associated with, so you don't see the “Microsoft Campaigns Against Counterfeit Software” subheadline in the Related Stories.

In a dynamic environment, the list of articles returned by this query can change as articles are added to the site.

## Text-Only Versions

Creating a text-only version of a web page (for printing it) is very easy to implement with CS-Direct. Using the CS-Direct rendering model, you can override a template that displays a given asset just by changing the page name used to render it. You do not need to pass the override template as a separate parameter. Creating a printer-friendly version of a page is simply a matter of adding a hyperlink that uses the plain-text version of the appropriate template.

For example, if you are pointing to an article:

```
http://myserver/servlet/
ContentServer?pagename=BurlingtonFinancial/Article/
Full&cid=987654321
```

You can change to the text-only version of the page by pointing to the text version of the template:

```
http://myserver/servlet/
ContentServer?pagename=BurlingtonFinancial/Article/
FullText&cid=987654321
```

Burlington Financial uses the convention of adding “Text” to the end of a Template asset name to indicate different styles of templates and elements. Some examples:

- Web format: `BurlingtonFinancial/Article/Summary`
- Plain text: `BurlingtonFinancial/Article/SummaryText`
- Web format: `BurlingtonFinancial/Page/SectionFront`
- Plain text: `BurlingtonFinancial/Page/SectionFrontText`

## Plain Text Parallel Site

In most web sites, the text-only pages have hyperlinks that take you back to the full web format pages. Or there are simply no navigable links on the printer friendly page. Burlington Financial's templates, though, are designed to show the visitor an entire plain text version of the site.

After you switch to the plain text version, you can continue to navigate around the plain text pages. However, not every single page in the Burlington Financial site is represented in the plain text version of the site. And the plain text pages do not have all the same content or hyperlinks as their graphics-rich versions. This was done intentionally, as a plain-text visitor would probably prefer a less complex version of a site. Extending Burlington Financial to include other parallel styles (WAP templates, WebTV, PDF, XML) would be very straightforward.

## E-mail This Story

Another very common feature on web sites is the ability to e-mail a story. This feature is relatively straightforward in Burlington Financial.

**Note**

Before you can email users, you have to configure Content Server and Content Centre to use the email feature.

First, in the `futuretense.ini` file, set these two properties:

```
cs.emailreturnto=<your email address>
cs.emailhost=po-1.XXXX.com (or the emailhost is)
```

Second, set this property in `futuretense_xcel.ini`:

```
xcelerate.emailnotification=true
```

The `BurlingtonFinancial/Util/EmailFront` and `BurlingtonFinancial/Util/EmailPost` elements call the article pagelet `BurlingtonFinancial/Article/Summary` to display the story in summary form. A more robust version of this code would check to make sure that the visitor entered a valid e-mail address before submitting the form. A real site would also keep records of which stories have been e-mailed, the sender's e-mail address, and the recipient's e-mail address.

## AssetMaker Asset Types

AssetMaker is a CS-Direct utility for constructing basic asset types. Two sample AssetMaker asset types are included in Burlington Financial: ImageFile and Stylesheet. These asset types use standard elements from AssetMaker without modification. Both have file upload fields for storing files in the database.

Burlington Financial includes JavaScript at the top of every page to do client-side browser detection and then load one of four corresponding Stylesheet assets. The element `BurlingtonFinancial/Common/SetHTMLHeader`, called at the top of each full-page template, uses the CS-Direct element `GetBlobURL` to get four different BlobServer URLs, one for each of the four different Stylesheet assets used by Burlington Financial. The actual `.css` file from the Stylesheet asset is served via the BlobServer, even though it isn't binary data.

## Mimetype

Both the imagefile and stylesheet asset types are served to the browser using the BlobServer servlet. To ensure that the browser knows how to handle arbitrary chunks of content, a mimetype code is saved and passed to the browser. CS-Direct includes a MimeType table for storing these codes.

The AssetMaker also allows asset types to define their own mimetype fields. Both the ImageFile and Stylesheet asset types include mimetype fields as part of their asset descriptor files. You can add your own mimetype codes and extensions to the MimeType table using Content Server Explorer or some other database tool.

## Collections of Collections

The collection asset type is how CS-Direct arranges content into manageable groups. Burlington Financial also demonstrates the use of a collection whose child asset type is a collection. This allows editors to easily rearrange groups of content on a web page simply by re-ranking a collection. These sub-collections can be used to build a library of favorite stories, breaking news, hot topics, etc.

In Burlington Financial, a named asset association called "HomeStoryGroups" was created from the page asset type to the collection asset type. The page template SectionFront looks for a collection in the StoryGroups slot. It forces the collection to be displayed using the BurlingtonFinancial/Collection/StoryGroups template:

```
<ASSET.CHILDREN NAME="SectionFrontPage" LIST="StoryGroups"
CODE="HomeStoryGroups"/
<IF COND="IsList.StoryGroups=true"
<THEN
<RENDER.SATELLITEPAGE PAGENAME="BurlingtonFinancial/Collection/
StoryGroups"
         ARGS_cid="StoryGroups.oid"
         ARGS_p="Variables.asset:id"/
</THEN
</IF
```

The StoryGroups template then loops over each collection in the asset:

```
<LOOP LIST="theGroups"
<RENDER.SATELLITEPAGE PAGENAME="BurlingtonFinancial/Collection/
PlainList"
         ARGS_cid="theGroups.oid"
         ARGS_p="Variables.p"/
<img src="/futuretense_cs/bf/images/dot_rule_125.gif"
width="125" height="1"/<P/
</LOOP
```

This can be seen on the news page. There is a collection called NewsGroup that is associated with the news page. This collection contains one child collection, called Energy List, which itself contains three articles. This lets an editor add or remove items from the News page by re-ranking the NewsGroup collection.

## Membership

The Burlington Financial sample site includes a membership component that has elements to sign up new members and log in existing members. These visitor registration elements are not robust enough for use on a real-world web site, but can give you a starting point for your own designs. For example, Burlington Financial has sample visitor account screens, allowing visitors to register and set their own preferences, but does not use this information to restrict visitor access to certain web pages, or to make recommendations based on a member's profile.

For more information about visitor registration in Burlington Financial and about security in general, see Chapter 31, "User Management on the Delivery System."

## Wire Feed

Both the home page template and the section front page template include a list of stories called “From the Wires.” This represents content that flows automatically onto the site. Large sites often subscribe to wire feed services or other content aggregators. Stories from these sources are moved onto a site with little human intervention.

Each of the page assets that make up the major sections of Burlington Financial are associated with a query asset. For the wire feed section, this query asset contains a query to look for article assets whose source field is set to WireFeed. The queries also look at the category field of each article. Each section in Burlington Financial contains certain categories of stories, so the wire feed queries try to match those categories.

The page asset named WireFeed contains a query to return wire feed stories regardless of their category.

## Featured Funds

The fund section front page template includes a list of funds called “Featured Funds.” This list contains funds that are selected using a Engage segment asset. Segment assets divide visitors into groups based on common characteristics.

You build a segment asset by first creating visitor data assets. A visitor data asset stores a single piece of information about visitors to the web site; a zip code, for example. Segments are built by selecting visitor data assets to base them on, and then setting qualifying values for those criteria. For example, you can create a zip code segment that uses the value in the zip code visitor data asset to display advertisements for local businesses.

The Featured Funds list displays funds based upon whether the web site visitor belongs to the `BFfrequentvisitor` segment or the `Highriskinvestor` segment.

## Fund Finder

Fund Finder is a form that allows you to search for funds based on the criteria that you select. Some of the Fund Finder form dropdowns are hard-coded into the form; the Fund Families that the form searches, however, are listed dynamically, using the CS-Direct Advantage concept of assetsets and searchstates.

A **searchstate** is a set of search constraints based on the attribute values

```
<SEARCHSTATE.CREATE NAME="ss"/>
<ASSETSET.SETSEARCHEDASSETS NAME="as"
ASSETTYPES="ProductGroups" CONSTRAINT="ss"/>
<ASSETSET.GETATTRIBUTEVALUES NAME="as"
TYPENAME="PAttributes" ATTRIBUTE="FundFamily"
LISTVARNAME="fflist"/>

<P><SELECT name="FundFamily" SIZE="3" MULTIPLE="1">
<OPTION SELECTED="" VALUE="NoPreference"/>No Preference
<LOOP LIST="fflist">
<OPTION/><csvar NAME="fflist.value"/>
</LOOP>
 </SELECT></P>
```

## Page Cache Parameters

By default, CS-Direct sets the `cacheinfo` property to `cs.pgcachefolder,*` for any `SiteCatalog` page entries that it creates when a you save a Template asset. However, there are times when you may not want pages to be cached.

Pages like `BurlingtonFinancial/Util/LoginPost` and `BurlingtonFinancial/Page/AccountAccess` are specifically set to `cs.nevercache`. This is necessary since they are visitor-specific, and you don't want one visitor to see another visitor's cached page results. Real customer sites need cache fine tuning for their pages.

CS-Direct has a set of default page criteria for creating SiteCatalog entries. You can also add additional page criteria variables, but the defaults should not be removed. For more information about caching and page criteria, see Chapter 5, “Page Design and Caching.”

Part 5

# Management System Features

This part describes how to customize certain features in the Content Server user interface on your Content Server management system.

It contains the following chapters:


- Chapter 34, “Customizing the User Interface”
- Chapter 35, “Coding for the InSite Editor”
- Chapter 36, “Customizing Workflow”

Chapter 34

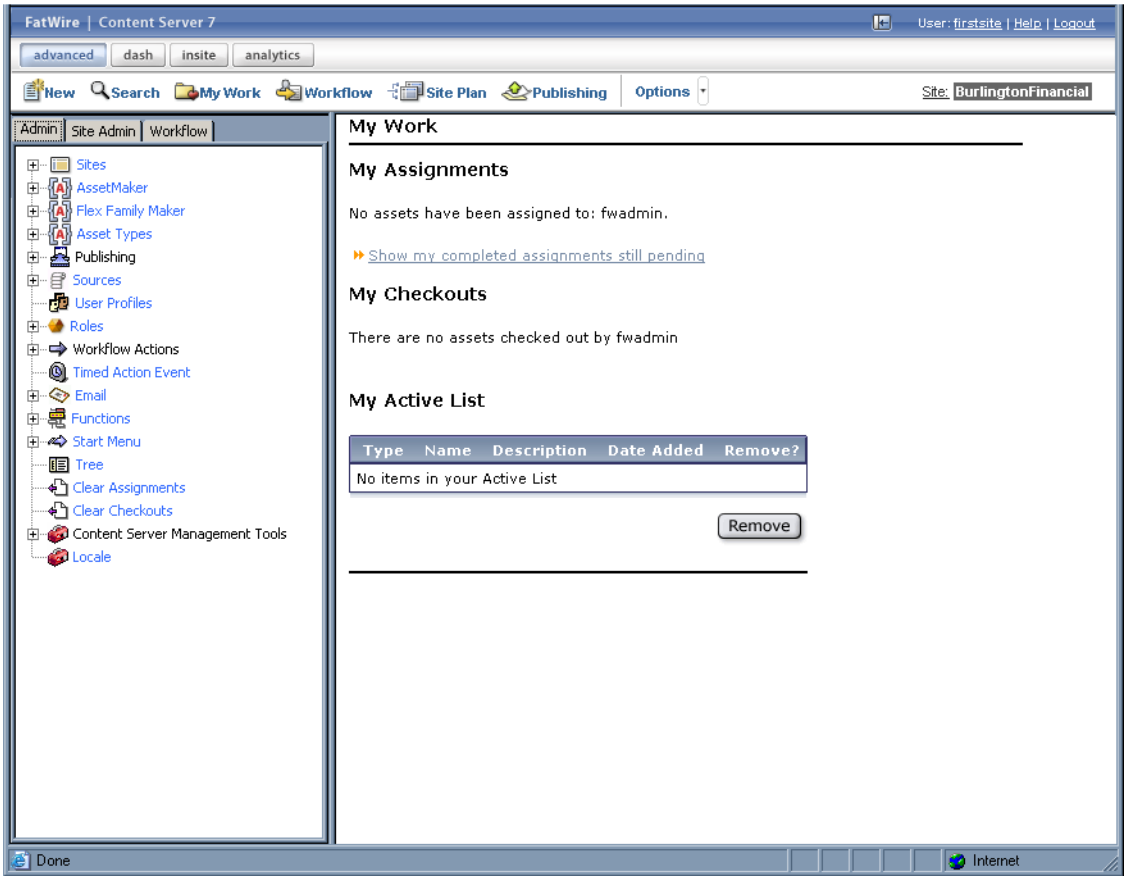
# Customizing the User Interface

Administrative and editorial users of Content Server interact with the product through various trees that display in the user interface. You can customize the Content Server user interface by modifying these trees. This chapter describes how to modify trees. It contains the following sections:

- Overview of the Tree
- Trees and Security
- Tree Error Logging

# Overview of the Tree

The tree appears as a set of tabs in the left pane of the main window of the Content Server interface, as shown in the following illustration:




Content Server tree tabs are created by the tree applet. You can create or modify your own trees by setting various parameters that will be passed to the tree applet. The tree applet accepts several kinds of parameters:

- Applet-wide parameters, which control the overall appearance and behavior of the applet
- Tree-specific parameters, which control the appearance and behavior of the tree
- Node parameters, which control the appearance and behavior of individual nodes on the tree
- OpURL Node parameters, which allow the tree to communicate with Content Server

A set of tree tab tables in the database stores information about tree configuration, including tab names, what roles have access to a tab, and the path to the element that populates the tree tab with data. You enter information into these tables via the Tree Tabs screens, which are accessed by clicking the Tree node on the **Admin** Tab.

## Loading the Tree Tabs

For most of the default tree tabs supplied with Content Server, requests for tree data pass through the `OpenMarket/Gator/UIFramework/LoadTab` element. The `LoadTab` element performs several basic tasks, such as checking for session timeout.

For example, the Product tab, found in the GE sample site that is provided with CS-Direct Advantage, completes the following steps as it loads:

1. Java code in the Product tab calls the `LoadTab` element.
2. The `LoadTab` element queries the TreeTab database tables to retrieve the elements that will load the data for the Product tree's top-level nodes. In this case, the elements are the `OpenMarket/Xcelerate/ProductGroups/LoadTree` element and the `OpenMarket/Xcelerate/Product/LoadTree` element.
3. The `OpenMarket/Xcelerate/ProductGroups/LoadTree` element and the `OpenMarket/Xcelerate/Product/LoadTree` element query the database for assets that correspond to the tree nodes and stream back node data to the tree applet.
4. The tree applet parses the node data and displays the nodes.
5. Java code in the Product tab calls an element to initialize its global pop-up menu, the `OpenMarket/Gator/UIFramework/LoadGlobalPopup` element. This element sends a `GetTypes` command to each tree loading element called by the Products tab. When the tree loading elements receive this command, they return a list of asset types whose start menu items that should appear in the global pop-up menu.
6. The `OpenMarket/Gator/UIFramework/LoadGlobalPopup` element finds the start menu items for the specified asset types and streams that information back to the tree.

Note that each asset type in the system must have a LoadTree element. The LoadTree element is a pointer to another element that actually loads the tree. If an asset type can have children, each of those children must have a LoadTree element. LoadTree elements have the following path:

`OpenMarket/Xcelrate/AssetType/`*`MyAssetType`*`/LoadTree`

where *`MyAssetType`* is the name of the asset type that the LoadTree element refers to.

LoadTree elements are called based on the asset type set in the `Section` field of the "Manage Tree" form.

Core asset types use one of several elements to load their trees. The following table contains a list of these elements:

| Asset Type | Location | Description |
|---|---|---|
| Flex Groups | `OpenMarket/Gator/UIFramework/LoadGroupNodes` | Displays a FlexGroup parent hierarchy and FlexAsset children |
| Flex Assets | `OpenMarket/Gator/UIFramework/LoadOrphanNodes` | Displays flex assets that do not belong to a flex group |
| Site Plan Tree | `OpenMarket/Xcelerate/AssetType/Page/LoadSiteTree` | Displays the SitePlan tree |

| Asset Type | Location | Description |
|---|---|---|
| Site Plan Associations | `OpenMarket/Gator/UIFramework/LoadChildren` | Displays asset associations in the SitePlan tree |
| Active List | `OpenMarket/Gator/UIFramework/LoadActiveList` | Displays the Active List tree |
| Administrative Tree | `OpenMarket/Gator/UIFramework/LoadAdminTree` | Displays the Administrative tree |
| Administrative Tree Helper Elements | `OpenMarket/Gator/UIFramework/Admin` | Loads helper elements for the Administrative tree |
| Asset Types | `OpenMarket/Gator/UIFramework/LoadAdministrationAsset` | Displays an asset type node at the top level of the tree and the names of all assets of that type on lower levels of the tree |

If you want to change the appearance or behavior of nodes in your tree, create a new tree loading element based on one of these standard elements. Your web site administrator can then specify the element's name and the path to that element in the Section Name and Element Name fields of the "New Tree" form, located off the "Tree Tabs" form. See the *Content Server Administrator's Guide* for more information about adding trees and the "New Tree" form.

See the "Node Parameters" section in this chapter for more information about modifying tree nodes.

## Applet-Wide Parameters

Applet-wide parameters are set in the `TreeAppletParams.xml` element. To modify the tree applet's behavior, change the parameter values set there, as shown in the following table:

**Table 14:** Applet-Wide Parameters

| Parameter | Description |
|---|---|
| `Debug` | Turns debugging on and off. Valid values are `true` and `false`. If `Debug` is set to `true`, Java console debug and error messaging is turned on. |
| `ServerBaseURL` | Sets the base string to which all the node data URL strings will be appended. For example, if the `ServerBaseURL` is set to `file://localhost`, and the value of the `LoadURL` parameter is `NodeReader.test`, the URL used for loading the tree's child nodes will be as follows: `file://localhost/NodeReader.test` |
| `BackgroundColor` | Sets the background color of the tree using a decimal RGB value. If this parameter is not set, the background color defaults to the color of the HTML frame in which the tree is embedded. |

**Table 14:** Applet-Wide Parameters

| Parameter | Description |
|---|---|
| `TotalPanes` | Sets the number of tree tabs that will be displayed. This value is set automatically. |
| `URLTarget` | The target frame in which to display node links. The default value is `XcelAction`—name of the pane on the right side of the browser window. |

## Tree-Specific Parameters

Tree-specific parameters are set by the "Add New Tree Tab" form and the `OpenMarket\Gator\UIFramework\TreeTabAdd.xml` element that creates the "Add New Tree Tab" form. To modify the tree's appearance or behavior, change the parameter values shown in the following table by using the using the form or by altering the `TreeTabAdd` element.

**Table 15:** Tree-Specific Parameters

| Parameter | Description |
|---|---|
| `Title` | Sets the text that is displayed on the tab.<br>This value is set in the `Title` field of the "Manage Tree" form, found on the `Admin` tab. |
| `ToolTip` | Sets the text that is displayed when the mouse pointer hovers over the tab index.<br>This value is set in the `Tool Tip` field of the "Manage Tree" form, found on the `Admin` tab. |
| `LoadURI` | The URI of the page to call to retrieve a node's children.<br>This value is set in the `TreeTabAdd` element. |
| `ActionURL` | The URL of the page that performs a pop-up menu action for a node in the tree. The default value points to the `OpURL.xml` element.<br>This value is set in the `TreeTabAdd` element. |
| `OpenIcon` | The path to the icon to use when depicting an expanded node. The default is a plus sign (`+`).<br>This value is set in the `TreeTabAdd` element. |
| `CloseIcon` | The path to the icon to use when depicting an unexpanded node. The default is a minus sign (`-`).<br>This value is set in the `TreeTabAdd` element. |
| `LineStyle` | Sets whether or not lines connect the nodes of the tree. Valid values are `Angled` and blank; `Angled` is the default. If the parameter is set to `Angled`, lines connect the nodes. If the value is left blank, no lines connect the nodes.<br>This value is set in the `TreeTabAdd` element. |
| `RootID` | Sets the ID of the root node. This string is used for specifying the node path. It defaults to the value of the `Title` parameter.<br>This value is set in the `TreeTabAdd` element. |
| `GlobalItems` | This value is set in the `GlobalItems` field of the "Manage Tree" form, found on the `Admin` tab. |
| `NodeItems` | This value is set in the `NodeItems` field of the "Manage Tree" form, found on the **Admin** tab. |

## Node Parameters

The node parameters determine the appearance and behavior of the nodes in your tree. To define the appearance and behavior of these nodes, you write an element which sets the node parameters (shown in the following table) and passes their values to the `BuildTreeNode.xml` element, which creates the tree nodes.

**Table 16:** Node Parameters

| Parameter | Description |
|---|---|
| `Label` | Specifies the text to be displayed for this node. The value does not have to be unique. Default is "". |
| `ID` | A string identifier that is unique within the tree, used by Content Server to express selection paths. The ID is specified by Content Server. |
| `ExecuteURL` | The URI value of the page to be displayed when completing the "`Execute`" action. This value will have the value of `ServerBaseURL` prepended to it.<br>If the node is not executable, do not include this parameter in the node data. |
| `URLTarget` | The frame target for `ExecuteURL`. If `ExecuteURL` is not included in the node data, it defaults to the target specified in the Applet-wide parameters. |
| `Description` | An alternative to the string specified in `Label`, if you choose this option on the tree-wide pop-up menu. The default value is "". |
| `Level` | The relative level of this node, represented by a number >= 0. A value of `0` indicates that the node is an immediate child of the node requesting the data.<br>To load more than one level of nodes at a time, set this value to a number greater than zero. The default value is `0`. |
| `Image` | The URI for the image to be prepended to the label. If this field is not included in the node data, no image will be displayed for that node. |

**Table 16:** Node Parameters

| Parameter | Description |
|---|---|
| `LoadURL` | The URI for the subtree hierarchy. If this field is not included in the node data, this node requires no additional loading.<br>The URL specified in this parameter must contain enough information so that the tree applet can find that node's children. For example, if your hierarchy is as follows:<br>Product Tab > Reebok > Running Shoes<br>the value of `LoadURL` is as follows:<br>`ContentServer?pagename=OpenMarket/Gator/UIFramework/LoadTab&AssetType=ProductGroups&populate=OpenMarket/Xcelerate/AssetType/ProductGroups/LoadTree&op=load&parent=Variables.parentid`<br>where "`parentid`" is the assetid of the "Running Shoes" asset, and "`op`" and "`populate`" are used by LoadTab to route to your tree load element. |
| `OKAction` | An action that will be displayed in the node's pop-up menu. This string may appear multiple times in the same node data set. |
| `OpURL` | The URL to execute a given action on the server. This value will be prepended with the value of the `ServerBaseURL` parameter.<br>Include this parameter in the node data unless the value of the `NodeItems` parameter is a null string, and thus has no `OKAction` specified. |
| `RefreshKeys` | Creates a key or set of set of keys which can be used to refresh the tree. Set the value to the ID of the current node. |

The following excerpt from the `LoadAdministrationAsset` element sets the values of the node parameters and passes those values to the `BuildTreeNode` element.

The `ListofAsset` list referred to in this excerpt is a list of information on assets of a given type. This list was generated by a SQL query that is executed elsewhere in the element.

```
<CALLELEMENT NAME="OpenMarket/Gator/UIFramework/BuildTreeNode">
   <ARGUMENT NAME="Label" VALUE="ListofAsset.name"/>
   <ARGUMENT NAME="Description" VALUE="ListofAsset.description"/>
   <ARGUMENT NAME="ID" VALUE="Variables.TreeNodeID"/>
   <ARGUMENT NAME="OpURL"
VALUE="ContentServer?pagename=OpenMarket/Gator/UIFramework/
TreeOpURL&#38;AssetType=Variables.AssetType"/>
   <ARGUMENT NAME="ExecuteURL"
VALUE="ContentServer?pagename=OpenMarket/Gator/UIFramework/
TreeOpURL&#38;AssetType=Variables.AssetType&#38;n0_=Variables.pack
edTreeNodeID&#38;op=displayNode"/>
   <ARGUMENT NAME="OKActions"
VALUE="Status;Inspect;Edit;Delete;refresh"/>
   <ARGUMENT NAME="Image" VALUE="Xcelerate/OMTree/TreeImages/
AssetTypes/Variables.AssetType.gif"/>
   <ARGUMENT NAME="RefreshKeys" VALUE="ListofAsset.id"/>
</CALLELEMENT>
```

To customize the appearance or behavior of tree nodes, copy one of the standard elements and modify the node arguments. Note that tree loading elements are passed the following variables, so any tree loading element that you create or customize must take these variables into account:

### Variables Passed in by the LoadTree element:

- `AssetType`, which is set to the section name that was created using the “New Tree” form
- `op`, which is set to `init`

### Variables Passed in by the LoadGlobalPopup element:

- `command`, which is set to `GetTypes`
- `AssetType`, which is set to the section name that was created using the “New Tree” form
- `varname`, which you set with a comma-separated list of asset types that you want to display start menu items for
- `popupvar`, which you set to either `true`, if you want to add items to the global pop-up, or `false`, if you do not need to add items to the pop-up

## Node Pop-up Commands

Each node on the tree has a menu that appears when the user right-clicks with the mouse. Commands on this menu allow you to refresh the node or load pages in the right side of the browser window. You can add commands to a node pop-up menu that allow you to load forms such as the status and publish forms. Any form that can be called using an asset type and ID is a good candidate for being called by a node pop-up command.

Add a new command to the node pop-up menu by completing the following steps:

1. Add the new command, exactly as you want it to appear, into the node's `OKActions` field.
2. Into the element that the node's `OpURL` refers to (usually the `TreeOpURL` element), add a new `IF` statement that calls the form you want to load.

   For example, the following code from the `TreeOpURL` element displays a node:

```
<IF COND="Variables.op=displayNode">
      <THEN>
      <callelement NAME="OpenMarket/Gator/UIFramework/
TreeIDFromPath">
      <argument NAME="TreePath" VALUE="Variables.TreeNodePath"/>
      </callelement>
         <setvar NAME="id" VALUE="Variables.ID"/>
         <callelement NAME="OpenMarket/Xcelerate/UIFramework/
ApplicationPage">
         <argument NAME="ThisPage" VALUE="ContentDetailsFront"/>
         <argument NAME="contentfunctions" VALUE="true"/>
         <argument NAME="AssetType"
VALUE="Variables.AssetType"/>
      </callelement>
      </THEN>
```

## Refreshing the Tree

Elements that can alter the tree are responsible for refreshing the tree so that it displays current data. There are three different types of refresh action that you can specify:

- Self, which refreshes the children of the specified node
- Parent, which refreshes the specified node and its children
- Root, which refreshes the entire tree

There are two steps to refreshing the tree:

1. Code your tree customization elements so that the tree nodes that you wish to refresh have `RefreshKeys`. `RefreshKeys` are keys—usually the asset ID of the current node—which allow the refresh to take place.
2. Call the `OpenMarket/Xcelerate/UIFramework/UpdateTreeOMTree` element, and pass the element the `_TreeRefreshKeys_` variable, specifying the type of refresh you want in the variable value.

You set the `RefreshKeys` for a node by passing the `RefreshKeys` argument to the `BuildTreeNode` element, as shown in the code sample in the "Node Parameters" section of this chapter.

To refresh the tree, call the `OpenMarket/Xcelerate/UIFramework/UpdateTreeOMTree` element, as shown in the following example:

```
<CALLELEMENT NAME="OpenMarket/Xcelerate/UIFramework/
UpdateTreeOMTree">
   <ARGUMENT NAME= "_TreeRefreshKeys_" VALUE= "Root:ActiveList"/>
</CALLELEMENT>
```

# Trees and Security

You can specify which users can see a tree by using the Content Server tree user interface. For more information about setting who can view a tree, see the *Content Server Administrator's Guide*.

# Tree Error Logging

All tree-related error and debug messages are logged to the Java Console. You can turn debugging on and off by supplying a value for the `Debug` parameter when you create a tree.

Note that enabling debug affects performance, so error logging should generally be turned off on the delivery system.

Chapter 35

# Coding for the InSite Editor

The InSite Editor is a CS-Direct feature that enables infrequent users to find, edit, and submit content directly from the rendered (Preview) version of an asset, which means that they do not have to learn how to use the Content Server interface.

Enabling this feature requires two general steps:

- Setting the `xcelerate.enableinsite` property in the `futuretense_xcel.ini` file to `true`.
- Coding templates that invoke the InSite Editor feature for the fields that you want content providers to be able to edit in this way.

This chapter describes how to code templates that invoke the InSite Editor. It contains the following sections:

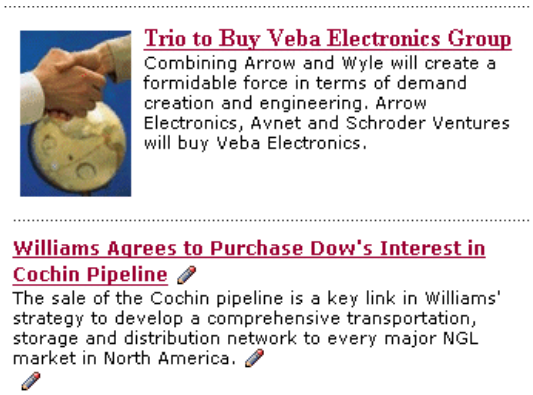

- Overview
- The INSITE.EDIT Tag
- Template Element Examples

# Overview

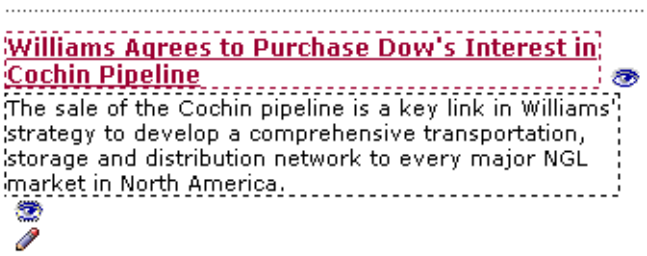
When a Content Server user with InSite Editor privileges previews any asset on a management system that has the InSite Editor configured, a separate control panel window appears, similar to the following:

The control panel provides a subset of the CS-Direct functions that you use to work with assets, including search and workflow functions. When a user selects an asset from the **Assignments** list in the control panel, CS-Direct displays it in its rendered (Preview) form in the browser window.

When you navigate to an asset that is rendered by a template that is coded for the InSite Editor, a blue pencil icon appears next to the asset. For example:



Then, when you click on the pencil icon, the icon changes shape and a dotted line surrounds the asset like this:




You can click in the area surrounded by the dotted line, make the necessary changes, and then click **Save** in the control panel. If the template renders more than one asset with InSite Editor fields—a collection template, for example—you can edit them all and they are all saved when you click the **Save** button.

Content providers can use the InSite Editor feature to edit and approve assets and finish assignments only if the following conditions are true:

- The `xcelerate.enableinsite` property in the `futuretense_xcel.ini` file is set to `true` on the Content Server system that they are working on.
- The value of `rendermode` is either `preview` or `live`. The InSite Editor does not appear when `rendermode=export`.
- The template or element rendering the asset is tagged correctly.
- The content providers have the appropriate edit privileges: they have the xceleditor ACL assigned to their user names and the assets they are working with are in a workflow state that gives them permission to edit those assets (or the assets are not in a workflow process at all).
- The asset was not created or edited with the CS-Desktop feature. That is, the asset has no value in its `externaldoctype` column.

For information about how to use the control panel, see the InSite Editor help file.

# The INSITE.EDIT Tag

To code your templates to enable the InSite Editor, you use one additional CS-Direct tag, the `INSITE.EDIT` tag. You use the tag in place of the `CSVAR` tag for the fields that you want users to be able to edit with the InSite Editor.

For example, this line of code displays the contents of an asset's description field:

```
<CSVAR NAME="Variables.description"/>
```

To have the description field rendered with the InSite Editor functionality enabled for it, you would change the line of code to this:

```
<INSITE.EDIT ASSETID="Variables.cid" ASSETFIELD="description"
   ASSETFIELDVALUE="Variables.description"
   ASSETTYPE="Variables.c"/>
```

When Content Server renders the template, it interprets that line of code as follows:

- On a management system with the InSite Editor enabled `(xcelerate.insiteewebedit=true)`, Content Server displays the contents of the description field with the blue pencil icon; the InSite Editor functionality is active.
- On the delivery system that the template is published to and that has the InSite Editor disabled (`xcelerate.insiteewebedit=false`), Content Server interprets the code as a `CSVAR` statement and displays it without the blue pencil icon.

In other words, you use the same template on both the management and the delivery system—Content Server knows what to do in each case.

There are three variations of the `INSITE.EDIT` tag which accept different parameters; for more information on these variations, see the *Content Server Tag Reference*.

Insite Editor can also handle fields that include embedded links. Use the `INSITE.EDIT` tag to display the field's contents.

## Parameters

The `INSITE.EDIT` tag takes the following parameters:

### ASSETID (required)

The ID of the asset whose field is to be displayed with the InSite Editor functionality. Typically, this value is held in the `cid` variable. For example:

```
ASSETID="Variables.cid"
```

### ASSETFIELD (required)

The name of the field that you want to display. You use different syntax for the value of this parameter when the asset identified by `ASSETID` is a basic asset than when it is a flex asset.

For example, to specify a field named `byline` for a basic asset, you use the following syntax:

```
ASSETFIELD="byline"
```

However, if the asset is a flex asset, the field is actually a flex attribute (which is also an asset). In this case, you use the following syntax:

```
ASSETFIELD="Attribute_byline"
```

### ASSETFIELDVALUE (required)

The current or default value of the field; that is, the value of the field when the asset is loaded. For example, this is the code for a field named `byline` for a basic asset:

```
ASSETFIELDVALUE="Variables.byline"
```

This is the code for an attribute named `byline` for a flex asset:

```
ASSETFIELDVALUE="Variables.Attribute_byline"
```

### ASSETTYPE (required)

The asset type of the asset identified with the `ASSETID` parameter.

### EWEBEDITPRO (optional)

If you are also using the eWebEditPro HTML editor from Ektron on your management system, you can use eWebEditPro as the input type for a field when it is in InSite Editor edit mode. By default, eWebEditPro is disabled.

To enable eWebEditPro for the field, provide this parameter and set it to `true`. For example:

```
EWEBEDITPRO=“true”
```

### WIDTH (optional)

By default, the width of the field in the InSite Editor edit mode is set to 100%. You can change the width by using either a percentage or a number in pixels.

### HEIGHT (optional)

By default, the height of a field displayed in the InSite Editor edit mode is 200 pixels. You can change the height by providing the number of pixels. Note that you cannot provide a percentage for the `HEIGHT` parameter.

## Syntax

Following is an example of the syntax for the `INSITE.EDIT` tag. This tag enables the InSite Editor for the `byline` field of a basic asset:

```
<INSITE.EDIT
   ASSETID="Variables.cid"
   ASSETFIELD="byline"
   ASSETFIELDVALUE="Variables.byline"
   ASSETTYPE=""Variables.c"
   WIDTH=“75”
   HEIGHT=“150”/>
```

If the asset is a flex asset, the same code would be written as follows:

```
<INSITE.EDIT
   ASSETID="Variables.cid"
   ASSETFIELD="byline"
   ASSETFIELDVALUE="Variables.Attribute_byline"
   ASSETTYPE=""Variables.c"
   WIDTH=“75”
   HEIGHT=“150”/>
```

## Supported Data Types and Input Types

This section describes the kinds of fields and attributes whose values can be edited with the InSite Editor.

### Basic Assets

For basic assets—the default CS-Direct asset types, the sample site article and imagefile asset types, and any asset type that you create with AssetMaker—you can use the InSite Editor on a field with any data type (`STORAGE TYPE`) other than timestamp. Those data types are as follows:

- `CHAR`
- `VARCHAR`
- `SMALLINT`
- `INTEGER`
- `BIGINT`
- `DOUBLE`

Additionally, the input type for the field must be compatible. That is, you should enable the InSite Editor functionality only for fields whose input style is text, textarea, or eWebEditPro.

### Flex Assets

For flex assets or flex parent assets, fields are flex attributes. You can use the InSite Editor to display the value of the flex attributes of any type other than `date` or `asset` in the template for a flex asset or flex parent asset.

Specifically, you can use the InSite Editor for attributes of any of the following types:

- `float`
- `integer`
- `money`
- `string`
- `text`
- `blob`

# Template Element Examples

This section provides a template example for both basic and flex assets. For longer examples, use Content Server Explorer to examine the following elements:

- `ElementCatalog/BurlingtonFinancial/Article/Full` (for a basic asset type)
- `ElementCatalog/BurlingtonFinancial/Article/Summary` (for a basic asset type)
- `ElementCatalog/OpenMarket/Demos/CatalogCentre/GE/Templates/blurb-story` (for a flex asset type)

## Example for Basic Asset

This template element is for a Burlington Financial article asset. It does the following:

- Logs the template as a dependent of the page or pagelet being rendered
- Loads the asset with an `ASSET.LOAD` tag, which logs the asset as an **exact** dependent of the page or pagelet being rendered
- Displays the `urlbody` field and the `description` field as fields that can be edited with the InSite Editor

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Article/BasicInsite
-
- INPUT
- Variables.c - asset type (Article)
- Variables.cid - id of the asset to display
- Variables.tid - template used to display the page(let)
- OUTPUT
-
-->

<!-- Log the template as a dependent of the pagelet being
rendered, so changes to the Template asset will force regeneration
of the page(let). -->

<IF COND="IsVariable.tid=true">
   <THEN>
   <RENDER.LOGDEP cid="Variables.tid" c="Template"/>
   </THEN>
</IF>

<!-- ASSET.LOAD logs an exact dependency between the asset and the
page being rendered with this element -->

<ASSET.LOAD NAME="anAsset" TYPE="Variables.c"
   OBJECTID="Variables.cid"/>

<!-- get all the primary table fields of the asset -->

<ASSET.SCATTER NAME="anAsset" PREFIX="asset"/>

<!-- Display the description and allow it to be edited through the
InSite Editor feature.-->

<INSITE.EDIT ASSETID="Variables.cid" ASSETFIELD="description"
   ASSETFIELDVALUE="Variables.asset:description"
   ASSETTYPE="Variables.c"/><br/>

<!-- Display the contents of the urlbody file and allow it to be
edited through the InSite Editor feature.-->
```

```
<ICS.GETVAR name="asset:urlbody" encoding="default"
output="bodyvar"/>
<INSITE.EDIT ASSETID="Variables.cid" ASSETFIELD="urlbody"
   ASSETFIELDVALUE="Variables.bodyvar" ASSETTYPE="Variables.c"/>

</FTCS>
```

Notice the line of code directly above the last `INSITE.EDIT` tag. When you use the `ASSET.SCATTER` tag to scatter a URL field, you must use the `ICS.GETVAR` method as shown in this example.

## Example for Flex Assets

This template element is for a product asset with a field named `productDescription`. It does the following:

- Logs the template as a dependent of the page or pagelet being rendered
- Creates an assetset with one product asset in it.
- Displays the `productDescription` field as a field that can be edited with the InSite Editor.

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Products/BasicInsite
-
- INPUT
- Variables.c - asset type (Products)
- Variables.cid - id of the asset to display
- Variables.tid - template used to display the page(let)
- OUTPUT
-
-->

<!-- Log the template as a dependent of the pagelet being
rendered, so
changes to the Template asset will force regeneration of the
page(let)
-->

<IF COND="IsVariable.tid=true">
   <THEN>
   <RENDER.LOGDEP cid="Variables.tid" c="Template"/>
   </THEN>
</IF>

<!-- Because this is a flex asset, we do not use ASSET.LOAD.
Instead, we create an assetset with the ASSETSET tag family and
name it 'as' -->

<ASSETSET.SETASSET NAME="as" ID="Variables.cid"
TYPE="Variables.c"/>
```

```
<!-- Retrieve the attribute named productDescription.-->

<ASSETSET.GETATTRIBUTEVALUES
   NAME="as"
   ATTRIBUTE="productDescription"
   TYPENAME="PAttributes"
   LISTVARNAME="productDescriptionList"/>

<!-- Display the productDescription and allow it to be edited
through the InSite Editor feature. Notice that for flex assets,
you prepend 'Attribute_' in front of the attribute name.-->

<INSITE.EDIT
   ASSETID="Variables.cid"
   ASSETFIELD="Attribute_productDescription"
   ASSETFIELDVALUE="productDescriptionList.value"
   ASSETTYPE="Variables.c"/><br/>

</FTCS>
```

## Example for an Attribute of Type Blob

This example retrieves and displays a Blob.

```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- get the URL -->

<ASSETSET.GETATTRIBUTEVALUES
       NAME="InSiteCourse"
       TYPENAME="BUAttribute"
       ATTRIBUTE="URL"
       LISTVARNAME="URLList"/>

<BLOBSERVICE.READDATA
       ID="URLList.value"
       LISTVARNAME="URLList"/>

URL:
<INSITE.EDIT
   ASSETID="Variables.cid"
   ASSETFIELD="Attribute_URL"
   ASSETFIELDVALUE="URLList.@urldata"
   ASSETTYPE="Variables.c"/><br/>
</FTCS>
```

Chapter 36

# Customizing Workflow

A Content Server workflow process is the series of states an asset moves through on its way to publication. The asset moves from one state to the next by taking a workflow step. Each step that the asset takes can be associated with a timed action, such as sending an e-mail to a user when an asset is assigned to them, or a workflow step condition, which prevents an asset from moving on to the next step if certain conditions are not fulfilled.

You must create the workflow step condition elements which specify the conditions that an asset must meet to move on to the next state, and the workflow action elements which perform various actions as the asset moves from one state to the next. This chapter describes these elements in greater detail and provide sample code for each element type. It contains the following sections:


- Workflow Step Conditions
- Workflow Actions

# Workflow Step Conditions

A workflow process is composed of one or more workflow states. Workflow steps move the asset from one workflow state to the next. Sometimes, however, there are conditions under which the asset should not move on to the next workflow state. You must create the element that defines the condition or conditions that prevent the asset from moving on to the next state.

This element receives the following data when it is called:

- An `IWorkflowable` object called `Object`, which represents the asset whose state is being changed
- An `IWorkflowStep` object called `Step`, which represents the current workflow step
- The `StepUser` variable, which contains the ID of the user attempting the step
- Variables specified as name-value pairs when a StepCondition is defined in the Content Server user interface. For more information about defining StepConditions, see the *Content Server Administrator's Guide*.

The workflow step condition element should check for a condition and return a Boolean value. If the value is false, the step will not proceed.

The following code comes from a sample workflow step condition element:

```
1   <?xml version="1.0" ?>
2   <!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
3   <FTCS Version="1.1">
4   <!-- OpenMarket/Xcelerate/Actions/Workflow/StepConditions/
    ExampleStepCondition
5   -
6   - INPUT
7   -
8   - OUTPUT
9   -
10  -->
11  <csvar NAME="This step condition will check if step can be
    taken"/><br/>
```

Line 12 sets an empty `ReturnVal` variable. In lines 31 and 36, this variable will be set with the reasons why the step cannot proceed.

```
12    <setvar NAME="ReturnVal" VALUE="Variables.empty"/>
13
14    <!--change the value of ReturnVal to a non-empty string
    later on, if you want to stop the step -->  <!-- most of
    the stuff below are debugging statements and also show you
    some items available to you to set up a condition for
    stopping the step-->
```

Line 16 uses the `WORKFLOWABLEASSET.GETDISPLAYABLENAME` tag to get the name of the asset that is in workflow.

```
15       <!-- get asset -->
16       <WORKFLOWABLEOBJECT.GETDISPLAYABLENAME OBJECT="Object"
    VARNAME="assetdisplayablename"/>
17       Object:<csvar NAME="Variables.assetdisplayablename"/
    ><br/>
```

Line 19 creates a variable called `StepUser` which will contain the ID of the user attempting to take the step. Line 20 uses the `USERMANAGER.GETUSER` tag to load the user's ID into the `StepUser` variable. Line 22 uses the `CCUSER.GETNAME` tag to retrieve a human-readable user name, and line 23 uses the `csvar` tag to display that user name.

```
18        <!-- get userid -->
19        Userid: <csvar NAME="Variables.StepUser"/><br/>
20        <USERMANAGER.GETUSER OBJVARNAME="myUserObj"
   USER="Variables.StepUser"/>
21        <CCUSER.GETNAME NAME="myUserObj" VARNAME="uname"/>
22        Username: <csvar NAME="Variables.uname"/><br/>
```

Line 24 uses the `WORKFLOWSTEP.GETID` tag to get the ID of the current workflow step. The `WORKFLOWSTEP.GETNAME` tag, used in line 25, loads the step with the specified name.

```
23         <!-- getstep -->
24        <WORKFLOWSTEP.GETID NAME="Step" VARNAME="sid"/>
25        Stepid: <csvar NAME="Variables.sid "/>
26        <WORKFLOWSTEP.GETNAME NAME="Step" VARNAME="sname"/>
27        Stepname: <csvar NAME="Variables.sname"/><br/><br/>
```

Lines 28 through 40 define the conditions that will stop the change of step from taking place. The `forcestop` and `notalloweduser` variables that the conditionals check were set as arguments when the sample step condition was defined in the Content Server interface. In a real step condition, you would test for the condition of your choice here—seeing whether an article asset has an associated image, for example.

```
28          <!-- This is the actual condition to stop the
   step. The following is just an example. -->
29            <if COND="Variables.forcestop=true">
30                <then>
31                <setvar NAME="ReturnVal" VALUE="You can not
   take this step because forcestop=true"/>
32                </then>
33                <else>
34                  <if
   COND="Variables.uname=Variables.notalloweduser">
35                    <then>
36                      <setvar NAME="ReturnVal" VALUE="You
   are not allowed to take this step"/>
37                    </then>
38                  </if>
39                </else>
40            </if>
41 </FTCS>
```

# Workflow Actions

As an asset moves through workflow, it can trigger a **workflow action**. A workflow action can do anything from send an email to alert a user that he has a new asset to evaluate to breaking a deadlock after a specified period of time has elapsed. There are five types of workflow actions:

- Step actions, which are executed as part of a transition between workflow states.
- Timed actions, which are triggered by deadlines when the asset is in a given state, thus associating the asset with a specific assignment.
- Deadlock actions, which are executed when an asset needs a unanimous vote in order to move to the next state, but the voters differ on which step the asset should take. The deadlock action will be executed whenever users choose different steps for the asset to move to.
- Group deadlock actions, which are executed when the assets in a workflow group need a unanimous vote in order to move to the next state, but the voters choose different steps, creating a deadlock.
- Delegation actions, which are executed when an asset is delegated. The delegated asset remains in its current workflow state, but is assigned to a new user.

Your workflow administrator must first define workflow actions using the Content Server user interface. Then you must create the elements that accomplish these workflow actions. FatWire provides several sample workflow action definitions for you to look at. For more information about defining workflow actions, see the *Content Server Administrator's Guide*.

The following sections describe sample workflow action elements.

## Step Action Elements

A Step Action element receives the following data when it is called:

- A `WorkflowEngine` object called `WorkflowEngine`.
- An `ObjectTotal` variable, which represents the total number of assets whose state is being changed.
- An `IWorkflowable` object called `Object`*nnn*, which represents the assets whose state is being changed. *nnn* is a number between `0` and `ObjectTotal -1`.
- An `IWorkflowStep` object called `Step`, which represents the workflow step being considered.
- A `StepTargetUser` variable, which is a comma-separated list of the step's target users.
- A `StepUser` variable, which contains the ID of the user attempting the step.
- A `Group` variable, which contains the ID of the workflow group to which the assets belong (if you are using workflow groups).
- Any variables that your workflow administrator has created in the definition for this Step Action.

The following Step Action element approves assets for publish; most other Step Action elements send an e-mail to the assignees.

```
1  <?xml version="1.0" ?>
2  <!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
3  <FTCS Version="1.1">
4  <!-- OpenMarket/Xcelerate/Actions/Workflow/StepActions/
   ApproveForPublish
5  -
6  - INPUT
```

```
7  -     Variables.ObjectTotal - number of loaded
   workflowasset objects
8  -  Object[n] - loaded workflowasset objects, where n = 0 -
   Variables.ObjectTotal
9  -targets - one or more comma separated names of PubTargets
   for which to approve the asset
10 -
11 - OUTPUT
12 -
13 -->
14
15 <!-- This is an action element called by step actions
   ApproveForPublish-->
16 This step action element will approve an asset for
   publish.<br/>
```

Line 18 uses the `SETCOUNTER` tag to create a counter which keeps track of the number of assets to approve. Lines 19 through 25 use the `LOOP` tag to loop through the assets and retrieve the asset types and IDs.

```
17 <!-- get the id and assettype of the asset(s) to approve --
   >
18 <SETCOUNTER NAME="count" VALUE="0"/>
19 <LOOP COUNT="Variables.ObjectTotal">
20 <WORKFLOWASSET.GETASSETTYPE OBJECT="ObjectCounters.count"
   VARNAME="assettype"/>
21 <WORKFLOWASSET.GETASSETID OBJECT="ObjectCounters.count"
   VARNAME="assetid"/>
22 <SETVAR NAME="idCounters.count" VALUE="Variables.assetid"/>
23 <SETVAR NAME="typeCounters.count"
   VALUE="Variables.assettype"/>
24 <INCCOUNTER NAME="count" VALUE="1"/>
25 </LOOP>
```

Line 27 uses the `STRINGLIST` tag to create a comma-separated list of publish target names. Lines 31 through 46 loop through this list, using the `PUBTARGET.LOAD` and `PUBTARGET.GET` tags to load information about the publish targets from the `PubTarget` table. This information and information about the assets to be approved are passed to the `ApprovePost` element for further processing in line 37.

```
26 <!-- approve for each destination -->
27 <STRINGLIST NAME="publishTargets" STR="Variables.targets"
   DELIM=","/>
28
29 <if COND="IsList.publishTargets=true">
30 <then>
31 <LOOP LIST="publishTargets">
32 <PUBTARGET.LOAD NAME="pubtgt" FIELD="name"
   VALUE="publishTargets.ITEM"/>
33 <if COND="IsError.Variables.errno=false">
34 <then>
35 Approving for publish to <CSVAR NAME="publishTargets.ITEM"/
   ><br/>
36 <PUBTARGET.GET NAME="pubtgt" FIELD="id" OUTPUT="pubtgt:id"/
   >
```

```
37 <CALLELEMENT NAME="OpenMarket/Xcelerate/PrologActions/
   ApprovePost">
38 <ARGUMENT NAME="targetid" VALUE="Variables.pubtgt:id"/>
39 <ARGUMENT NAME="assetTotal" VALUE="Counters.count"/>
40 </CALLELEMENT>
41 </then>
42 <else>
43 Cannot approve for publish to destination: <CSVAR
   NAME="publishTargets.ITEM"/>, Error: <CSVAR
   NAME="Variables.errno"/>
44 </else>
45 </if>
46 </LOOP>
47 </then>
48 <else>
49 Cannot approve for publish. This step action requires a
   targets argument with one or more comma separated
   publishing destination names.
50 </else>
51 </if>
52
53 </FTCS>
```

## Timed Action Elements

Timed Action elements receive the following data when they are called:

- A `WorkflowEngine` object called `WorkflowEngine`.
- A `WorkflowAssignmentTotal` variable, which contains the total number of assignments for which this action applies.
- An `IWorkflowAssignment` object called `WorkflowAssignment`*nnn*, which represents assignments to apply the action to. *nnn* is a number greater than zero.
- An optional `Group` variable, which contains the ID of the workflow group to which the assets belong (if you are using workflow groups)
- Any variables that your workflow administrator has created in the definition for this Timed Action.

The following excerpt is from a Timed Action element that sends an e-mail. The text of the subject and body of this e-mail are set in the Workflow E-mail forms that you access from the Admin tab in the Content Server user interface. The body text expects the following variables:

- `Variables.assetname`, which contains the name of the current asset
- `Variables.assigner`, which is the name of the user who completed the previous state in the workflow process
- `Variables.instruction`, which is the text that the assigner puts in the `Action to Take` text box as he or she completes an assignment

```
1  <!-- This is a timed action element -->
2
3  <!-- get total assignments -->
4  <if COND="IsVariable.WorkflowAssignmentTotal=true">
```

```
5   <then>
6   <setvar NAME="NumOfAssignments"
    VALUE="Variables.WorkflowAssignmentTotal"/>
7   </then>
8   <else>
9   <setvar NAME="NumOfAssignments" VALUE="0"/>
10  </else>
11  </if>
12
13  <!-- For each assignment object, get assignee -->
14  <setcounter NAME="COUNT" VALUE="0"/>
15  <if COND="Variables.NumOfAssignments!=0">
16  <then>
17  <loop FROM="0" COUNT="Variables.NumOfAssignments">
18  <setvar NAME="tmp"
    VALUE="WorkflowAssignmentCounters.COUNT"/>
19  <WORKFLOWASSIGNMENT.GETASSIGNEDUSERID NAME="Variables.tmp"
    VARNAME="assigneduserid"/>
20
21  <!-- get user -->
22                  <WORKFLOWASSIGNMENT.GETASSIGNEDOBJECT
    NAME="Variables.tmp" OBJVARNAME="assignedobj"/>
23
24  <!-- get asset -->
25  <WORKFLOWABLEOBJECT.GETDISPLAYABLENAME OBJECT="assignedobj"
    VARNAME="assetname"/>
26
27  <!-- get deadline and format it -->
28  <WORKFLOWASSIGNMENT.GETDEADLINE NAME="Variables.tmp"
    VARNAME="deadline"/>
29  <DATE.DEFAULTTZ VARNAME="tzone"/>
30  <DATE.CLOCKLIST LISTVARNAME="DueTime"
    CLOCK="Variables.deadline" TIMEZONE="Variables.tzone"/>
31  <setvar NAME="time" VALUE="DueTime.fulldate
    DueTime.longtime"/>
32
33  <!-- get email address --->
34  <USERMANAGER.GETUSER USER="Variables.assigneduserid"
    OBJVARNAME="userobj"/>
35  <CCUSER.GETNAME NAME="userobj"
    VARNAME="assigned_user_name"/>
36  <CCUSER.GETEMAIL NAME="userobj" VARNAME="EmailAddress"/>
37
38  <IF COND="IsVariable.EmailAddress=true">
39  <THEN>
40
41  <!-- load email object -->
42  <EMAILMANAGER.LOAD NAME="Variables.emailname"
    OBJVARNAME="emailobject"/>
43
```

In lines 45 and 48, the variables in the e-mail object, `subject` and `body`, are replaced by their values.

```
44 <!-- translate subject -->
45 <EMAIL.TRANSLATESUBJECT NAME="emailobject"
   PARAMS="assetname=Variables.assetname" VARNAME="subject"/>
46
47 <!-- translate body -->
48 <EMAIL.TRANSLATEBODY NAME="emailobject"
   PARAMS="assetname=Variables.assetname&#38;time=Variables.ti
   me" VARNAME="body"/>
49
50 <!-- send mail -->
51 <sendmail TO="Variables.EmailAddress"
   SUBJECT="Variables.subject" BODY="Variables.body"/>
52 </THEN>
53 <ELSE>
54 Email address: None<br/>
55 </ELSE>
56 </IF>
57
58 <inccounter NAME="COUNT" VALUE="1"/>
59 </loop>
60 </then>
61 </if>
62
63
64 </FTCS>
```

## Deadlock Action Elements

Deadlock Action elements receive the following data when they are called:

- A `WorkflowEngine` object
- An `ObjectTotal` variable, which represents the total number of deadlocked assets
- An `IWorkflowable` object called `Objectnnn`, which represents the deadlocked assets
- An `IWorkflowStep` object called `Step`, which represents the workflow step
- A `StepTotal` variable, which contains the number of steps chosen by individual users
- A `StepUser` variable, which contains the ID of the user attempting the step
- An optional `Group` variable, which contains the ID of the workflow group to which the assets belong (if you are using workflow groups)
- Any variables that your workflow administrator has created in the definition for this Deadlock Action.

The following Deadlock Action element sends an e-mail to the users who approve the asset.

The text of the subject and body of this e-mail are set in the Workflow E-mail forms in the Content Server administrative user interface. The body text expects the following variables:

- `Variables.assetname`, which contains the name of the current asset

- `Variables.header` and `Variables.message`, which contain the text of the e-mail's body

```
1  <?xml version="1.0" ?>
2  <!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
3  <FTCS Version="1.1">
4  <!-- OpenMarket/Xcelerate/Actions/Workflow/DeadlockActions/
   SendEmailToAssignees
5  -
6  - INPUT
7  -
8  - OUTPUT
9  -
10 -->
11
12 <!-- This is an action element called by step actions
   SendAssignmentEmail and SendRejectionEmail-->
13
14 <csvar NAME="This deadlock action element will send
   emails"/><br/>
```

Line 16 uses the `EMAILMANAGER.LOAD` tag to load an e-mail object.

```
15 <!-- load email object -->
16 <EMAILMANAGER.LOAD NAME="Variables.emailname"
   OBJVARNAME="emailobject"/>
```

Lines 17 through 25 create a `NumOfSteps` variable, which contains either the total number of assets being delegated or zero.

```
17 <!-- get total steps -->
18 <if COND="IsVariable.StepTotal=true">
19 <then>
20 <setvar NAME="NumOfSteps" VALUE="Variables.StepTotal"/>
21 </then>
22 <else>
23 <setvar NAME="NumOfSteps" VALUE="0"/>
24 </else>
25 </if>
26
27 <removevar NAME="Step"/>
28 <setvar NAME="Header" VALUE="The following users have
   chosen the corresponding steps that has resulted in a
   deadlock. Please take appropriate actions to resolve
   deadlock:"/>
29 <setvar NAME="Message" VALUE="Variables.empty"/>
```

Lines 30 through 75 loop through the list of users who have put the asset in deadlock, creating an e-mail for each one. Line 39 uses the `USERMANAGER.GETUSER` tag to load the user information of the user specified in the ID. Lines 40 and 41 use `CCUSER` tags to get the user's name and e-mail address.

```
30 <!-- For each assignment object, get asignee -->
31 <setcounter NAME="COUNT" VALUE="0"/>
32 <if COND="Variables.NumOfSteps!=0">
33 <then>
```

```
34 <loop FROM="0" COUNT="Variables.NumOfSteps">
35 <!-- get assigner -->
36
37 <setvar NAME="userid"
   VALUE="Variables.StepUserCounters.COUNT"/>
38 <!-- get email address --->
39 <USERMANAGER.GETUSER USER="Variables.userid"
   OBJVARNAME="userobj"/>
40 <CCUSER.GETNAME NAME="userobj" VARNAME="user_name"/>
41 <CCUSER.GETEMAIL NAME="userobj" VARNAME="EmailAddress"/>
```

Lines 42 through 47 use the `WORKFLOWSTEP` and `WORKFLOWSTATE` tags to retrieve the asset's starting and ending steps and states.

```
42 <WORKFLOWSTEP.GETNAME NAME="StepCounters.COUNT"
   VARNAME="stepname"/>
43 <WORKFLOWSTEP.GETSTARTSTATE NAME="StepCounters.COUNT"
   VARNAME="startstate"/>
44 <WORKFLOWSTEP.GETENDSTATE NAME="StepCounters.COUNT"
   VARNAME="endstate"/>
45
46 <WORKFLOWSTATE.GETSTATENAME NAME="Variables.startstate"
   VARNAME="startstatename"/>
47 <WORKFLOWSTATE.GETSTATENAME NAME="Variables.endstate"
   VARNAME="endstatename"/>
48
49
50 <setvar NAME="Message" VALUE="Variables.Message
   Variables.user_name: Variables.stepname - "/>
51 <!--
52 user:<csvar NAME="Variables.user_name"/><br/>
53 step name:<csvar NAME="Variables.stepname"/><br/>
54 startstate name:<csvar NAME="Variables.startstate"/><br/>
55 endstate name:<csvar NAME="Variables.endstate"/><br/>
56 -->
57
58 <!-- get asset -->
59 <WORKFLOWABLEOBJECT.GETDISPLAYABLENAME
   NAME="Variables.ObjectCounters.COUNT" VARNAME="assetname"/>
```

In lines 62 and 65, the variables in the e-mail object, `subject` and `body`, are replaced by their values.

```
60 <!-- translate subject -->
61 <SETVAR NAME="params"
   VALUE="username=Variables.user_name&#38;header=Variables.He
   ader&#38;message=Variables.Message&#38;assetname=Variables.
   assetname"/>
62 <EMAIL.TRANSLATESUBJECT NAME="emailobject"
   PARAMS="Variables.params" VARNAME="subject"/>
63
64 <!-- translate body -->
65 <EMAIL.TRANSLATEBODY NAME="emailobject"
   PARAMS="Variables.params" VARNAME="body"/>
66
```

```
67 <!-- send mail -->
68 <sendmail TO="Variables.EmailAddress"
   SUBJECT="Variables.subject" BODY="Variables.body"/>
69
70 <inccounter NAME="COUNT" VALUE="1"/>
71 </loop>
72 </then>
73 </if>
74 email message:<csvar NAME="Variables.Header
   Variables.Message"/><br/>
75
76
77 </FTCS>
```

## Group Deadlock Action Elements

Group Deadlock action elements receive the following data when they are called:

- A `WorkflowEngine` object called `WorkflowEngine`.
- An `ObjectTotal` variable, which represents the total number of deadlocked assets.
- An `IWorkflowable` object called `Object`*nnn*, which represents the deadlocked asset. *nnn* is a number greater than zero.
- An `IWorkflowStep` object called `Step`, which represents the workflow step.
- A `StepTotal` variable, which contains the number of steps chosen by individual users
- A `StepUser` variable, which contains the ID of the user attempting the step.
- A `Group` variable, which contains the ID of the workflow group that is deadlocked.
- Any variables that your workflow administrator has created in the definition for this Group Deadlock Action.

The following Group Deadlock Action element sends an e-mail to the users who approve the asset.

The text of the subject and body of this e-mail are set in the Workflow E-mail forms in the Content Server administrative user interface. The body text expects the following variables:

- `Variables.assetname`, which contains the name of the current asset
- `Variables.header` and `Variables.message`, which contain the text of the e-mail's body

```
1  <?xml version="1.0" ?>
2  <!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
3  <FTCS Version="1.1">
4  <!-- OpenMarket/Xcelerate/Actions/Workflow/GroupActions/
   SendEmailToAssignees
5  -
6  - INPUT
7  -
8  - OUTPUT
9  -
10 -->
```

```
11
12 <!-- user code goes here -->
13
14 <csvar NAME="This group deadlock action element will send
   emails"/><br/>
15 <!-- load email object -->
16 <EMAILMANAGER.LOAD NAME="Variables.emailname"
   OBJVARNAME="emailobject"/>
17 <!-- get group -->
18 <WORKFLOWENGINE.GETGROUPID ID="Variables.Group"
   OBJVARNAME="grpobj"/>
19 <WORKFLOWGROUP.GETNAME NAME="grpobj" VARNAME="GroupName"/>
20
21 <!-- get total steps -->
22 <if COND="IsVariable.StepTotal=true">
23 <then>
24 <setvar NAME="NumOfSteps" VALUE="Variables.StepTotal"/>
25 </then>
26 <else>
27 <setvar NAME="NumOfSteps" VALUE="0"/>
28 </else>
29 </if>
30
31 <removevar NAME="Step"/>
32 <setvar NAME="Header" VALUE="The following users have
   chosen the corresponding steps that has resulted in a
   deadlock for the group: Variables.GroupName. Please take
   appropriate actions to resolve deadlock:"/>
33 <setvar NAME="Message" VALUE="Variables.empty"/>
34 <!-- For each assignment object, get asignee -->
35 <setcounter NAME="COUNT" VALUE="0"/>
36 <if COND="Variables.NumOfSteps!=0">
37 <then>
38 <loop FROM="0" COUNT="Variables.NumOfSteps">
39 <!-- get assigner -->
40 <setvar NAME="userid"
   VALUE="Variables.StepUserCounters.COUNT"/>
41 <!-- get email address --->
42 <USERMANAGER.GETUSER USER="Variables.userid"
   OBJVARNAME="userobj"/>
43 <CCUSER.GETNAME NAME="userobj" VARNAME="user_name"/>
44 <CCUSER.GETEMAIL NAME="userobj" VARNAME="EmailAddress"/>
45
46 <WORKFLOWSTEP.GETNAME NAME="StepCounters.COUNT"
   VARNAME="stepname"/>
47 <WORKFLOWSTEP.GETSTARTSTATE NAME="StepCounters.COUNT"
   VARNAME="startstate"/>
48 <WORKFLOWSTEP.GETENDSTATE NAME="StepCounters.COUNT"
   VARNAME="endstate"/>
49
50 <WORKFLOWSTATE.GETSTATENAME NAME="Variables.startstate"
   VARNAME="startstatename"/>
```

```
51 <WORKFLOWSTATE.GETSTATENAME NAME="Variables.endstate"
   VARNAME="endstatename"/>
52
53 <!-- get asset -->
54 <WORKFLOWABLEOBJECT.GETDISPLAYABLENAME
   NAME="Variables.ObjectCounters.COUNT" VARNAME="assetname"/>
55
56 <!-- set message -->
57 <setvar NAME="Message" VALUE="Variables.Message Asset:
   Variables.assetname, User: Variables.user_name, Step:
   Variables.stepname -- "/>
58
59 <!-- translate subject -->
60 <SETVAR NAME="params"
   VALUE="username=Variables.user_name&#38;header=Variables.He
   ader&#38;message=Variables.Message&#38;assetname=Variables.
   assetname"/>
61 <EMAIL.TRANSLATESUBJECT NAME="emailobject"
   PARAMS="Variables.params" VARNAME="subject"/>
62
63 <!-- translate body -->
64 <EMAIL.TRANSLATEBODY NAME="emailobject"
   PARAMS="Variables.params" VARNAME="body"/>
65
66 <!-- send mail -->
67 <sendmail TO="Variables.EmailAddress"
   SUBJECT="Variables.subject" BODY="Variables.body"/>
68
69 <inccounter NAME="COUNT" VALUE="1"/>
70 </loop>
71 </then>
72 </if>
73 email message:<csvar NAME="Variables.Header
   Variables.Message"/><br/>
74
75 </FTCS>
```

## Delegation Action Elements

Delegation action elements receive the following data when they are called:

- A `WorkflowEngine` object called `WorkflowEngine`.
- A `CurrentUser` variable, which contains the ID of the user who is delegating the asset.
- An optional `Group` variable, which contains the ID of the workflow group. All objects to be delegated must be in the same workflow group.
- An `ObjectTotal` variable, which represents the total number of assets being delegated.
- An `IWorkflowable` object called `Object`*nnn*, which represents the assets being delegated. *nnn* represents a number greater than zero.

Delegation action elements should be coded like other Workflow Action elements.

Part 6

# Web Services

This part explains how to integrate Content Server with any client application that has a SOAP interface.

It contains the following chapters:


- Chapter 37, “Overview of Web Services”
- Chapter 38, “Creating and Consuming Web Services”

Chapter 37

# Overview of Web Services

This chapter introduces web services and explains how they work with Content Server. It describes what is supplied.

This chapter contains the following sections:


- What Are Web Services?
- SOAP and Web Services
- Supported SOAP Version
- Supported WSDL Version
- Related Programming Technologies


As a standard part of the CS-Direct product, web services require no additional installation or configuration.

# What Are Web Services?

Content Server enables you to create, deploy, and publish your own web services, as well as consume web services from other applications. Web services, which are a collection of operations that are accessible via standard XML messaging over the Internet, enable data exchange independent of the programming language, operating system, or hardware used by a given target system. With regard to Content Server, web services provide a standard means to expose content and functionality for consumption by remote applications, including ERP (enterprise resource planning), CRM (customer relationship management), and commerce systems.

# SOAP and Web Services

Integration between Content Server and other applications is accomplished by transforming data using HTTP and XML data formats. The key XML format for web services is SOAP (Simple Object Access Protocol). SOAP is a W3C specification that extends HTTP to enable distributed applications to make remote-procedure calls (RPCs) over the Internet. As a language, SOAP defines the XML elements used to describe the parameters, return values, and so on required for RPC-style interactions. SOAP messages are transmitted point-to-point and handled in request-and-response fashion. Content Server, which supports SOAP, can exchange data with any application that has a SOAP interface. When processing SOAP requests, Content Server leverages its native support for XML and its efficient page-evaluation and delivery mechanisms.

**Note**

Content Server uses the JWSDP (Sun) implementation for web services. If the application server where Content Server is installed also has its own web services implementations, there will be conflicts with the SOAP `MessageFactory`. To resolve the conflicts, you must set a default `MessageFactory` via VM parameters before starting the application server. With the default set, Content Server can determine which `MessageFactory` to use for Content Server web services.

For example:

JBoss comes with its own web services. If you wish to use JBoss's `MessageFactory` for other reasons and also use Content Server web services, you must set the default `MessageFactory` to that of JBoss as follows:

```
-Djavax.xml.soap.MessageFactory=org.jboss.ws.core.
   soap.MessageFactoryImpl
```

If you need only Content Server web services, set the default `MessageFactory` as follows:

```
-Djavax.xml.soap.MessageFactory=com.sun.xml.messaging.
   saaj.soap.ver1_1.SOAPMessageFactory1_1Impl
```

# Supported SOAP Version

Content Server supports SOAP 1.1. You will need to know the capabilities and limitations of the SOAP protocol to write web services for Content Server. SOAP standard syntax is described in detail at the W3C web site:

`http://www.w3.org`

# Supported WSDL Version

WSDL (web services description language) is an XML format that describes distributed services on the Internet. A WSDL file describes the location of the service and the data to be passed in messages for particular operations. With regard to Content Server, these messages contain procedure-oriented information.

Content Server supports WSDL 1.1. You will need to understand the web services description language to write web services for Content Server and use the predefined WSDL files shipped with Content Server. The SOAP standard syntax is described in detail at the following W3C web site:

`http://www.w3.org/TR/wsdl.html`

# Related Programming Technologies

To write web services for Content Server, you should have a basic understanding of some or all of the following related technologies:

- XML
- SOAP
- WSDL
- JSP
- Java
- J2EE (Java 2)
- .Net

Chapter 38

# Creating and Consuming Web Services

This chapter explains how to integrate Content Server with remote applications over the Internet using web services protocols. In the context of the Content Server development framework, it teaches you how to create and consume basic web services.

This chapter contains the following sections:


- Using Predefined Web Services
- Creating Custom Web Services

# Using Predefined Web Services

Content Server includes a complete array of asset-delivery functions implemented as web services. These services can be accessed by any technology that can produce a web-services-enabled client. Supplied web-service capabilities are comparable to existing Content Server APIs (XML, JSP, Java, and COM).

Each supplied service is represented by a predefined WSDL (web services description language) file that contains descriptions of multiple web-services operations. Individual WSDL files define the interface and methods for web-services operations that correspond to Content Server functions. Related operations are grouped and collectively described according to function.

The WSDL file is used to generate the code required to interact with Content Server from a client application. You can generate client code automatically using various third-party applications that read WSDL files, or manually by examining the WSDL description and writing the client code from scratch. The resulting client stub constitutes a suitable interface for interaction with Content Server. When executed, the code creates a SOAP request based on the WSDL operation.

Most times you will have control over the client interaction with Content Server. For access by potentially unknown client applications, however, the supplied WSDL files can also be posted to a URL and registered via UDDI (universal description, discovery, integration) for remote access.

## Accessible Information

Any web services client that supports SOAP and follows the predefined WSDL specifications can access the following information from the Content Server database:

- Site map of a content management site
- Blob data, such as a PDF file
- List of all the valid asset types and asset subtypes at a content management site
- List of assets that match specified search criteria
- Metadata associated with a particular asset

## WSDL File Location

Predefined WSDL files for Content Server are automatically installed with the CS-Direct application in the following location:

`http://`*`install_dir`*`/futuretense_cs/Xcelerate/wsdl/*.wsdl`

## Process Flow for Predefined Web Services

The following general steps describe how a request from a web services client program is processed using a supplied WSDL file:

1. The supplied WSDL file includes a description of the format for the request (input data expected by Content Server) and the format for the return data. The WSDL file maps standard data types for applications written in Java, Visual Basic, or other programming languages to XML schema data types.

2. The client program uses instructions in the WSDL to transform data from an input source (for example, a structured file) to an XML schema that is consistent with what the Content Server web service interface expects.
3. The client generates a SOAP envelope that includes the required data and transmits it to the content management site.
4. Content Server receives the SOAP message.
5. An XML parser and transformation utility map the data in the SOAP message to the format required by Content Server.
6. Content Server invokes the appropriate CS-Direct seed classes.
7. Seeds invoke the specified Content Server action.
8. Content Server returns requested data (name/value pairs) in the output format defined by the WSDL file to the client application.
9. The SOAP processor for the client application maps the XML schema data types to native data types for the specific programming language used.

## Consider Your Data for Predefined Web Services

Data for the predefined WSDL operations is passed using RPC-style interactions (versus exchanging entire XML documents) to your program. These are mostly strings, but Content Server also includes classes that handle native objects with complex data types; for example, SearchStates and ILists.

## Generating the Client Interface

Use the WSDL files to generate an interface for your client application. A variety of tools that generate client code from WSDL files are available. These tools support output for different programming languages. Choose a tool that produces code in the target language for your client application, and run it on the WSDL file that describes the operations you need. The resulting client stub makes all Content Server operations available to your client program.

## Writing Client Calls

The code generated from the WSDL file provides an interface to Content Server functions. Once available, you can call the functions from your application, as needed.

# Creating Custom Web Services

With Content Server, you can create web services that map data from any Content Server functions that you want to expose. Because of its support for XML, Java, and JSP, the existing Content Server development environment provides a familiar platform for developing web services. A supplied tag set enables you to build a SOAP response and stream SOAP encapsulated data to and from applications. As with the prepackaged web services, the Content Server delivery capability and page-evaluation pipeline are used to process SOAP requests. For web services, the client is a program, not a browser.

To create a custom web service, follow these general steps:

1. Consider Your Data for Custom Web Services
2. Creating a Content Server Page
3. Writing a Content Server Element
4. Creating a WSDL File

## Process Flow for Creating Custom Web Services

The following general steps describe how a request from a web services client program is processed:

1. The client program wraps whatever inputs are required in SOAP and passes them to Content Server.
2. The client uses instructions in the WSDL file to transform data from an input source (for example, a structured file) to an XML schema that is consistent with what the Content Server web service interface expects.
3. The client generates a SOAP envelope that includes the required data and transmits it to the content management site.
4. Content Server receives the SOAP message.
5. An XML parser and transformation utility map the data in the SOAP message to the format required by Content Server.
6. Content Server invokes the appropriate CS-Direct seed classes.
7. Seeds invoke the specified Content Server action.
8. Content Server returns requested data (name/value pairs) in the output format defined by the WSDL file to the client application.
9. SOAP processor for the client application maps the XML schema data types to native data types for the specific programming language used.

## Consider Your Data for Custom Web Services

Data is passed using RPC-style interactions (versus exchanging entire XML documents) to your program. Consider your data and verify that you will be dealing with simple XSI data types. Content Server supports all W3C XSI primitive data types without modification.

**Note**

Support for complex web-services data types is possible in Content Server but requires that you create your own Java classes and deploy them on the application server. If you plan to create your own data types, FatWire recommends that you consult with Content Server technical support before doing so.

## Creating a Content Server Page

Each web service requires page entry in the `SiteCatalog` table. The page entry to the SiteCatalog is a name that points to the element that calls the Content Server function

described by your web service. The SiteCatalog stores all valid entries for pages at your site, including those that invoke web services.

The page is invoked by a request from the client. In turn, the response from Content Server is encapsulated in SOAP and returned to the client. Remember that for web services the client is a program (instead of a browser), and the response is XML (instead of HTML).

#### To create a Content Server page for web services

1. Start Content Server Explorer and log in to Content Server. For instructions, refer to the online help or the instructions in this guide.
2. In the left pane, open the `SiteCatalog` table.
3. Select the folder for your site.
4. In the **pagename** field, create an entry for the last part of the page name for your web service in the `SiteCatalog` table.
5. In the **root element** field, create a root entry for your web service. (Including the SiteCatalog entry in the SiteCatalog root avoids a table lookup and ensures that the element name of the first child element is mapped to the specified pagename.)
6. In either of the **resargs** fields, add the following optional arguments, if appropriate:
   - `cs.session=false` bypasses application server session management for the life of the SOAP request without using existing session objects or creating new session objects on behalf of the current request. This improves performance by reducing the application server load for native requests and requests from clients that do not require session persistence. Although supported on any page, the `cs.session=false` resarg is mainly intended for use with SOAP services.
   - `cs.contenttype=text/xml` prepares the root element for processing. Specifically, it causes the XML engine to properly respect namespace on tags and prevents default HTML compression from occurring. This is required only if you expect the request to come through a browser. Unless the web services request is always received as a SOAP request, you must include this resarg in each SiteCatalog entry to override the HTML compression. Provided that your input XML is well formed, you can be sure that the content output will be proper XML.
7. In the **csstatus** field, enter `Live`, or the appropriate status at this time.
8. Choose **File > Save All**, or click the **Save All** toolbar button to save your work.

## Writing a Content Server Element

The Content Server element contains the code for the function you want to expose. The element handles data and formats the SOAP response. To format the SOAP response, include the SOAP XML tags supplied with Content Server in your code. Content Server automatically generates the XML for the SOAP envelope.

Content Server elements written for web services in XML and JSP must not contain extra whitespace or comments because, unlike HTML, XML and its SOAP implementation have stricter parsing requirements. Because XML and JSP pages are handled the same way as HTML pages and are not filtered by Content Server, extra whitespace or comments can corrupt the SOAP message. Comments (XML or JSP) should appear only after the `soap.message` tag.

### To write an element for a web service

1. Start Content Server Explorer and log in to Content Server (if you have not already done so). For instructions, refer to the online help or this guide.
2. In the left pane, select and highlight the `ElementCatalog` table.
3. Create a new folder in the `ElementCatalog` table: choose **File > New Folder**.
4. Select and highlight the new folder, and right-click anywhere in the right pane and select **New** from the pop-up menu.

   A new row appears in the table.
5. In the **elementname** field, enter an appropriate name for your web service as the name of the element.
6. In the **description** field, enter a short description of the element.
7. Click in the **url** field, and click the button that appears.

   The **New File** dialog box appears.
8. In the **Type/Ext** field, select **XML** or **JSP** as the file type from the drop-down list.
9. Click **OK**. Content Server Explorer opens its default editor. Content Server creates a file containing the skeleton code required of all XML or JSP elements. Enter your element code, including the required Content Server SOAP tags. For example, you can use the following code for XML:

```
<?xml version="1.0" ?><!DOCTYPE FTCS SYSTEM
"futuretense_cs.dtd">
<FTCS Version="1.2">
<!-- WebServices/helloworld
-
- INPUT
-
- OUTPUT--->
<soap.message  ns="mynamespace">
      <soap.body tagname="HelloWorldOut">
      <echoStringOut xsi:type="xsd:string">
      <csvar NAME="Variables.echoString"/>
         </echoStringOut>
      </soap.body>
</soap.message>

</FTCS>
```

10. Choose **File > Save All** to save your work.

**Note**

Choosing **File > Save** instead of **File > Save All** saves your file, but does not make it available to the application server.

# Creating a WSDL File

WSDL (web services description language) files describe the web service so that a basic web services client can be automatically generated based on information it contains. If you are not using the predefined services provided with Content Server, you can optionally create your own WSDL file to describe your web service.

A WSDL file includes the SOAP address, SOAP action, a description of the format for the request (input data expected by Content Server), and the format for the return data. The WSDL file maps standard data types for applications written in Java, Visual Basic, or other programming languages to XML schema data types. Use any of the predefined WSDL files for Content Server functions as a template to get started.

## Writing WSDL File Elements

A WSDL file has four sections:

- `Types` - specifies the data format and schema definition for operations. The type correlates with return data.
- `Message` - names inputs and outputs. This describes what the Content Server page must send back.
- `Operation` - describes inputs and outputs.
- `Binding` - specifies the SOAP action and operations.
- `Service` - describes associated port and binding with a URL.

In these sections, specify the following key XML elements:

- target namespace
- service name
- port name
- operation name
- input parameters (corresponding to simple data types) for the operation. Simple XSI data types (string, integer, float, and so on) return a single value.

## WSDL File Example

Each WSDL file is a collection of interrelated operations, logically grouped together according to their Content Server function. Completed web services return XML in the form of a SOAP encapsulated response. You will need to understand web services description language to write web services for Content Server. Content Server supports WSDL 1.1.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:s="http:/
/www.w3.org/2001/XMLSchema" xmlns:s0="http://FatWire.com/someuri/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
targetNamespace="http://FatWire.com/someuri/" xmlns="http://
schemas.xmlsoap.org/wsdl/">
   <types>
   </types>
```

```
    <message name="HelloWorldIn">
       <part name="echoString" type="s:string"/>
    </message>
    <message name="HelloWorldOut">
       <part name="echoStringOut" type="s:string"/>
    </message>
    <portType name="HelloWorldPortType">
       <operation name="helloworld">
          <documentation>FOR LATER</documentation>
          <input message="s0:HelloWorldIn"/>
          <output message="s0:HelloWorldOut"/>
       </operation>
    </portType>
    <binding name="HelloWorldBinding" type="s0:HelloWorldPortType">
       <soap:binding transport="http://schemas.xmlsoap.org/soap/
        http" style="rpc"/>
       <operation name="helloworld">
          <soap:operation soapAction="WebServices/" />
          <input>
             <soap:body use="encoded"/>
          </input>
          <output>
             <soap:body use="encoded"/>
          </output>
       </operation>
    </binding>
    <service name="HelloWorld">
       <port name="HelloWorldPort" binding="s0:HelloWorldBinding">
          <soap:address location="http://localhost:8080/servlet/
           ContentServer"/>
       </port>
    </service>
</definitions>
```

Part 7

# Engage

This part describes how to use Engage to design an online site that gathers information about your site visitors and customers and to then use that information to personalize the information that is displayed for each visitor.

It contains the following chapters:


- Chapter 39, “Creating Visitor Data Assets”
- Chapter 40, “Recommendation Assets”
- Chapter 41, “Coding Engage Pages”
- Chapter 42, “Memory-Centric Visitor Tracking”

Chapter 39

# Creating Visitor Data Assets

Engage lets you design online sites that gather information about your site visitors and customers, and then to use that information to personalize the product placements and promotional offerings that are displayed for each visitor.

Customizing your online sites begins with visitor data. The definitions of visitor data types are treated as assets in the Content Server database. There are three kinds of visitor data assets: visitor attributes, history attributes, and history types.

This chapter describes the visitor data assets and presents procedures for creating them. It contains the following sections:


- About Visitor Data Assets
- Creating Visitor Data Assets
- Verifying Your Visitor Data Assets
- Approving Visitor Data Assets

# About Visitor Data Assets

You create visitor data assets so that you can use them to group your site visitors into segments. There are three kinds:

- Visitor attributes
- History attributes
- History types

When you create visitor data assets, you create entries in the visitor data tables in the Content Server database and you reserve a place in the database to store information of that kind for your site visitors.

## Visitor Attributes

Visitor attributes hold types of information that specify one characteristic only (scalar values). For example, you can create visitor attributes named “years of experience,” “job description,” or “number of children.”

When the visitor changes the data, the new data overwrites the old data. Engage does not assign a timestamp to the data that is stored as a visitor attribute and does not store revisions. For example, if a visitor changes his entry for “job description” from “butcher” to “baker,” the information that the visitor was once a butcher is overwritten. You cannot, for example, create a segment based on bakers who used to be butchers.

For historical data, you must use history types.

## History Attributes and History Definitions

**History attributes** are individual information types that you group together to create a vector of information that Engage treats as a single record. This vector of data is the **history definition**. For example, a history definition called “purchases” can consist of the history attributes “SKU,” “itemname,” “quantity,” and “price.”

Engage references data stored as a history definition as a whole or an aggregate. It assigns a timestamp to each instance of the recorded definition and keeps each of those records. This means that you can sum or count history definitions and you can determine the first time or the last time a history definition was recorded for a visitor. Using the example in the preceding paragraph, you can create a segment based on the amount of money a visitor spends on specific items during a set period of time.

History definitions store historical data.

## Segments

Segments are assets that divide visitors into groups based on common characteristics. Segments are built by determining which visitor data assets to base them on and then setting qualifying values for those criteria.

When you create visitor data assets, you create fields. These fields can be used in two places:

- As criteria for segments. That is, as configuration options in the Engage Segment Filtering forms (because you define segments with the visitor data assets). In other words, the choices you make about the data types for the attributes determine their

appearance and behavior in the Segment forms. When you create these assets, you are customizing the Segment forms.

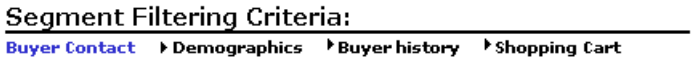
- On your public site pages. That is, as form fields or hidden fields on registration pages and other pages.

Segments are the key to personalizing merchandising messages with Engage. When visitors browse your site, the information they submit is used to qualify them for segment membership. When the site displays a page with a recommendation or promotion, Engage determines which segments a visitor belongs to and displays the product recommendations or promotional messages that are designated for those segments.

For help with creating segments, see the *Content Server Advanced Interface User's Guide*.

## Categories

Engage uses categories to group visitor attributes and history definitions into useful links on the Segment forms. The visitor data assets are listed under categories that are displayed across the top of the forms. For example:




Because visitor attributes and history definitions are such different types of assets, you must use separate categories for them. You create categories when you enter text in the Category field on a visitor attribute or history definition form. If the name you enter is not in use yet, Engage creates a new category.

## Developing Visitor Data Assets: Process Overview

There are five general steps for creating and using visitor data assets (fields):

1. A cross-functional design team including developers and marketers determines what kind of data you want to gather about your site visitors.
2. You (the developers) create and define the necessary visitor attributes, history attributes, and history definitions by using the forms in Engage.
3. The marketers use the Segment Filtering forms in Engage to categorize groups of visitors based on these visitor attributes, history attributes, and history definitions.
4. You program the appropriate site pages with the Engage XML or JSP object methods to collect and store the data, using either server-side validation or Javascript to validate the input on the pages. For example, you can create an online registration form for visitors to fill out by using JavaScript to validate the input and the Engage XML or JSP tags to process and store that information in the Content Server database.
5. When visitors browse your site, the information they submit is used to qualify them for segment membership. If your site is using promotions and recommendations based on segments, the message displayed for the visitor is personalized based on the segments that he or she qualifies for.

# Creating Visitor Data Assets

Before you begin creating visitor data assets, be sure that you have completed the following tasks:

- Met with the marketing and design teams to determine the kinds of data that you want to collect about visitors.
- Examined the Segment Filtering forms so that you understand the context in which the visitor data assets that you create are used by the marketers. Additionally, note that the **visitor data assets** are listed by their **descriptions** rather than by their names in the Segment Filtering forms.

You can use the following data definitions for your visitor and history attributes:

- string – can hold up to 255 characters
- boolean – true and false are the only legal values
- short – valid range of values is 0 through 255
- integer – valid range of values is 0 through 65,535
- long – valid range of values is 0 through 65,535
- double – valid range of values is 0 through 4,294,967,295
- date – format is `yyyy-mm-dd hh:mm:ss.s`
- money – format is currency; valid range of values is unlimited
- binary – for visitor attributes only; used for binary data such as image files or cart objects

**Note**

Binary visitor attributes can record binary data for individual visitors. Visitor attributes of this type are not displayed in the Segment Filtering forms and cannot be used to define a segment. Creating an attribute of type binary reserves space in the Content Server database that you use to store objects by using the XML object method `VDM.SAVESCALAROBJECT` or its JSP equivalent `vdm:savescalarobject` to convert an object from the Content Server name space into a binary form.

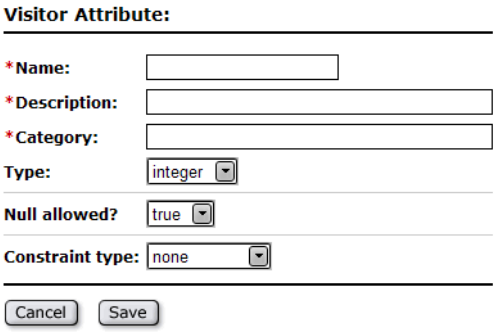
## Creating Visitor Attributes

Use the procedures in this section to create visitor attributes with the Engage forms.

### Step 1: Name and Define the Visitor Attribute

1. If necessary, log in to the Content Server interface, and if given a choice, select a site.
2. Click **New** on the button bar.
3. Select **New Visitor Attribute** from the list of asset types. (Visitor Attribute asset types must be enabled for your site.)

The “Visitor Attribute” form appears.




**Note**

If **Visitor Attribute** does not appear in the menu list, it means that your login password combination does not give you administrator rights. Contact the site administrator and request that the admin user profile be assigned to your user name.

4. In the **Name** field, enter a unique, descriptive name for the attribute (field). You can enter up to 32 alphanumeric characters, including spaces. The first character must be a letter.
5. In the **Description** field, enter a description of the attribute (field). Enter a value (alphanumeric characters) that will help you easily identify the attribute (attributes are listed by their descriptions rather than their names in the **Segment Filtering** forms).
6. In the **Category** field enter the category for the attribute. The text that you enter in this field determines where the attribute is listed in the Segment Filtering form. You can enter up to 32 alphanumeric characters.

**Note**

Categories for visitor attributes must be different from the categories for history definitions.

## Step 2: Configure the Data Type

1. In the **Type** field select a data type.
2. If you selected **string**, in the **Length** field enter the maximum number of characters allowed for input in the attribute (field). You can enter a value up to 255.
3. In the **Null allowed** field, select **true** to allow null values or **false** to require input for the attribute when it is used. For example, an attribute with a Boolean data type cannot allow a null value.
4. If you selected **false** in the **Null allowed** field, in the **Default Value** field enter a default value that is appropriate for the attribute’s data type. For example, if the data type is “integer” the default value must be a number.

**Note**

If you selected **binary** as the data type, you cannot specify a default value for the attribute.

## Step 3: Configure the Constraint Criteria

The constraint options that are available for validating input into the attribute depend on the data type that you designated for the attribute.

### Option 1: Configure the attribute to accept free-form text

In the **Constraint type** field select none from the drop-down list. For example, a visitor attribute named “residence” and of type string might accept unconstrained text as input.

### Option 2: Configure the attribute to accept input from a range of values

To configure the attribute to accept a specific range of values, the data type must be integer, short, long, double, or money.

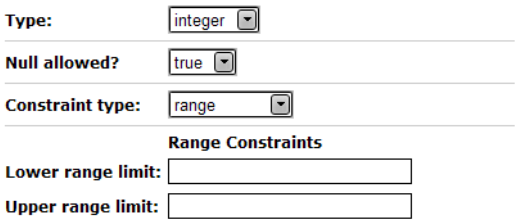
1. In the **Constraint type** field and select range.

   The form displays range fields.

2. In the **Lower range limit** field and specify the smallest possible value that can be accepted in the attribute when it is used as a field. This value cannot be a negative number.

3. In the **Upper range limit** field, enter the largest possible value that can be accepted in the attribute when it is used as a field. (For a short data type, you can enter a value up to 255; for integer, up to 65,535; for double, up to 4,294,967,295; for money, unlimited.)

   For example, an attribute named “age” can be restricted to accept only values between 1 and 110.




### Option 3: Configure the attribute to offer a set list of values in a drop-down list

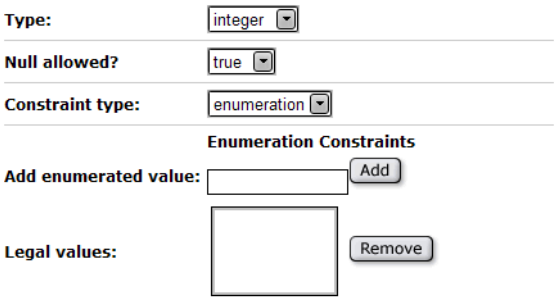
1. In the **Constraint type field** and select enumeration.

   The form displays text boxes for adding options.

2. In the **Add Enumerated Value** field, enter the name of the first option. For example, an attribute named “gender” can have “female” as an option.

3. Click **Add**.

The option is moved to the list.




4. Repeat these steps for each of the options that you want to make available for this attribute (field).

### Step 4: Save the Attribute

1. (Optional) If more than one site is set up and you have access to those sites, specify whether you want to share this attribute with the other sites. In the **more...** drop-down menu on the "Inspect" screen, select **Share Visitor Attribute**.
2. When you are finished configuring the visitor attribute, click **Save**.

   Engage creates an entry for this attribute in the visitor data asset tables in the Content Server database and reserves a place in the database to store information of that type for your site visitors.

   Engage then displays a summary of the attribute in the "Inspect" form.

You can now use this visitor attribute in a segment.

**Note**

After a visitor attribute is used to define a segment, deleting the attribute invalidates the segment. Be sure to correct your segments if you delete an attribute.

## Creating History Attributes

The purpose of history attributes is different from the purpose of visitor attributes: you create history attributes to be used by history definitions. You cannot use them in the Segment Filtering forms until they are used to define a history definition.

Use the procedures in this section to create history attributes by using the Engage forms.

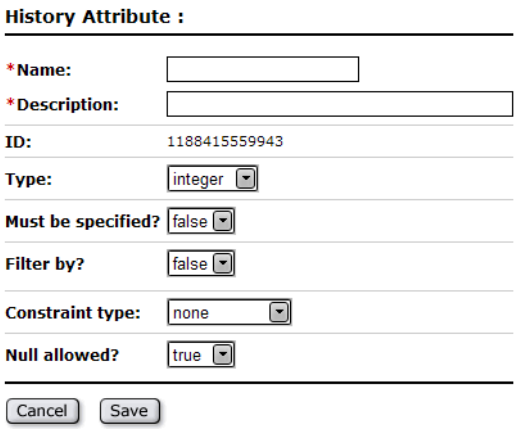
**Note**

You cannot edit or delete a history attribute after it has been used to define a history definition. You also cannot remove it from the history definition. If you must change a history attribute after it has been used to define a history definition, it is best to stop using the history definition. Create a new history attribute, create a new history definition, and then start using the new history definition.

## Step 1: Name and Define the History Attribute

1. If necessary, log in to the Content Server interface, and if given a choice, select a site.
2. Click **New** on the button bar and select **History Attribute** from the list.

   The "History Attribute" form appears.




> **Note**
>
> If History Attribute does not appear in the menu list, it means that your login/password combination does not give you administrator rights. Contact the site administrator and request that the admin user profile be assigned to your user name.

3. In the **Name** field, enter a unique, descriptive name for the attribute (field). You can enter up to 32 alphanumeric characters, including spaces. The first character must be a letter.
4. In the **Description** field, enter a description of the attribute (field). Enter a value (alphanumeric characters) that will help you easily identify the attribute (attributes are listed by their descriptions rather than their names in the **Segment Filtering** forms).

## Step 2: Specify that the Attribute Can Be a Filter Criterion

1. If you want this attribute to be a required field when the history definitions that use it are used to define a segment, click in the **Must be specified** field and select **true**.
2. Click in the **Filter by** field and select **true**.

   If you do not set **Filter by** to true, the marketers cannot use the attribute (field) as a constraint for any history definition that it belongs to when they create segments.

   If the data type for this attribute is numeric, then by default the attribute is included in the list of attributes that can be selected for a Total constraint in a segment—whether you set **Filter by** to true or to false. However, if you want to use a numeric attribute as a constraint in any other way, you must set **Filter by** to true.

## Step 3: Configure the Data Type

1. Click in the **Type** field and select a data type.

2. If you selected **string**, in the **Length** field enter the maximum number of characters allowed for input in the attribute (field).
3. Click in the **Null allowed** field and select `true` to allow null values or `false` to require input for the attribute when it is used. For example, an attribute with a Boolean data type cannot allow a null value.
4. If you selected `false` in the **Null allowed** field, in the **Default Value** field enter a default value that is appropriate for the attribute's data type. For example, if the data type is "integer" the default value must be a number.

## Step 4: Configure the Constraint Criteria

The constraint options available for validating input into the attribute depend on the data type you designated for the attribute.

### Option 1: Configure the attribute to accept free-form text

Click in the **Constraint type** field and select none from the drop-down list. For example, a history attribute named "Street Name" and of type string might accept unconstrained text as input.

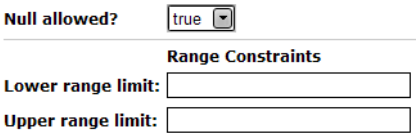
### Option 2: Configure the attribute to accept input from a range of values

To configure the attribute to accept a specific range of values, the data type must be integer, short, long, double, or money.

1. In the **Constraint type** field, select range.

   The form displays range fields.
2. In the **Lower range limit** field, specify the smallest possible value that can be accepted in the attribute when it is used as a field. This value cannot be a negative number.
3. In the **Upper range limit** field, enter the largest possible value that can be accepted in the attribute when it is used as a field. (For a short data type, you can enter a value up to 255; for integer, up to 65,535; for double, up to 4,294,967,295; for money, unlimited.)

   For example, an attribute named "Number of Items" can be restricted to accept only values between 1 and 50.



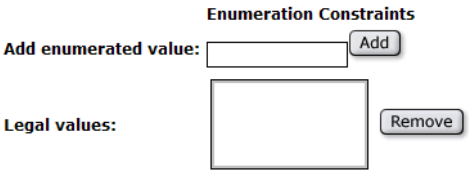
### Option 3: Configure the attribute to offer a drop-down list of specific values

1. In the **Constraint type field**, select enumeration.

   The form displays text boxes for adding options.

2. In the **Add Enumerated Value** field, enter the name of the first option. For example, an attribute named “Browser” can have “Netscape” as an option.

3. Click **Add**.




4. Repeat these steps for each of the options that you want to make available for this attribute (field).

## Step 5: Save the History Attribute

When you are finished configuring the history attribute, click **Save**.

Engage creates an entry for this attribute in the visitor data asset tables in the Content Server database and reserves a place in the database to store information of that type for your site visitors.

You can now use this history attribute to define a history definition.

# Creating History Definitions

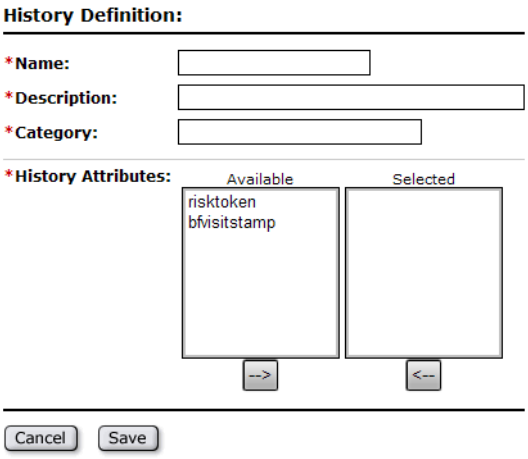
History definitions are made up of history attributes. Therefore, there must be at least one history attribute created before you can create a history definition.

Use this procedure to create history definitions by using the Engage forms:

1. If Engage is not open, log in.

2. Click **New** and select **History definition** from the list.

   The “History definition” form appears:



**Note**

If History definition does not appear in the menu list, it means that your login/password combination does not give you administrator rights. Contact the site administrator and request that the admin user profile be assigned to your user name.

3. In the **Name** field, enter a unique, descriptive name for the history definition (record). You can enter up to 32 alphanumeric characters, including spaces. The first character must be a letter.
4. In the **Description** field, enter a description of the history definition (record). Enter a value (alphanumeric characters) that will help you easily identify the history definition (history definitions are listed by their descriptions rather than their names in the **Segment Filtering** forms).
5. In the **Category** field, enter a category for the history definition. The text that you enter in this field determines how the history definition is sorted and displayed in the Segment Filtering forms. You can enter up to 32 alphanumeric characters.

   **Note**

   Categories for history definitions must be different from the categories for visitor attributes.

6. In the **Fields** area, select the history attributes that make up this history definition. Select an attribute and then click the right arrow to move it to the list on the right. Use control + click to select more than one attribute at the same time.

   **Note**

   After a history attribute is used to define a history definition, you can no longer edit or delete that history attribute.

7. Click **Save**.

   Engage creates an entry for this history definition (record) in the visitor data asset tables in the Content Server database and reserves a place in the database to store information of that type for your site visitors.

   Engage then displays a summary of the history definition in the “Inspect” form.

You can now use this history definition in a segment.

# Verifying Your Visitor Data Assets

To determine that you correctly set up your visitor attributes, history attributes, and history definitions, examine the Segment Filtering forms and decide whether the visitor assets that you created were configured correctly:

- Create segments that use each of the visitor attributes and history definitions that you created.
- Determine that the constraint definitions are correct and that the input ranges are accepting the correct range of input.

For help with creating segments, see the *Content Server Advanced Interface User's Guide*.

# Approving Visitor Data Assets

When your visitor data assets are ready, approve them so that they can be published to the delivery system.

When a history definition is published, the history attributes that are used to define it are also published. That means that you have to approve all the history attributes in a history definition before the history definition can be published.

To approve any asset, select **Approve for Publish** from the drop-down list in the icon bar in the asset's "Edit" or "Inspect" form.

The procedure for approving any asset, including visitor attributes, history attributes, and history definitions, is presented in the *Content Server Advanced Interface User's Guide*.

Chapter 40

# Recommendation Assets

**Recommendations** are assets that determine which products or content should be featured or "recommended" on a rendered page. These assets are a set of rules that might be based on the segments the visitors qualify for, and, in some cases, relationships between the product and/or content assets.

This chapter describes how recommendations work and how to create a custom element that returns assets to be recommended. It includes the following sections:


- Overview
- Creating a Dynamic List Element

# Overview

A recommendation asset collects, assesses, and sorts CS-Direct Advantage product and content assets and then recommends the most appropriate flex assets, such as assets for the current visitor. How does it determine which are the most appropriate assets? By consulting the list of segments that the visitor belongs to.

The CS-Direct Advantage product and content flex assets are rated for their importance to each segment. When a recommendation asset is called from a template, Engage determines which segments the current visitor qualifies for, and then selects the assets that are identified by the recommendation as having the highest rating for those segments. These are the assets that are "recommended" to the visitor.

There are three kinds of recommendations:

- **Static Lists** – return a static list of recommended items
- **Dynamic Lists** – return a list of recommended items that is generated by a dynamic list element that you create
- **Related Items** – return a list of recommended items based on relationships between flex assets, such as products.

Engage uses a recommendation's configuration options and the asset ratings to constrain the list when the list contains more items than the template is programmed to display. For related items recommendations, Engage also uses asset relationships to constrain the list. For all recommendations, Engage eliminates assets that are rated 0 for the current visitor.

The recommendation asset is the only Engage asset that can be assigned a template. You code your recommendation templates to render the items that the recommendation returns in an appropriate way on the rendered page.

The template tells the recommendation how many assets to return, and the recommendation asset determines which assets to select and return to the template based on the way it is configured and on the segments that the current visitor belongs to.

There are several XML and JSP object methods (tags) that you can use to code templates for recommendations. For information about coding templates when you are using Engage, see Chapter 41, "Coding Engage Pages." For information about all of the Engage tags see the *Content Server Tag Reference.*

## Development Process

Following are the basic steps for setting up recommendations:

1. Developers and designers meet with the marketing team to define all the merchandising messages that you want to display on your site and to plan how to represent those messages using recommendation and promotion assets.
2. The developers and designers use the XML or JSP object methods to design and code templates for the recommendations. Chapter 41, "Coding Engage Pages," explains how to code these templates.
3. If the web site will use dynamic list recommendations, the developers code the dynamic list elements that return the assets to recommend. The "Creating a Dynamic List Element" section of this chapter explains how to code dynamic list elements.

4. The marketing team uses the Engage recommendation wizard to create and then configure the recommendations. They assign the appropriate template to the appropriate recommendation.
5. Using the Engage product and content asset forms, the marketers rate how important the assets are to each segment, and, therefore, to the individual visitors who become members of those segments. (Typically, you assign ratings to flex parents, such as product parents, instead of to individual assets.)
6. For each "related items" recommendation, the marketers configure the relationships maintained by those recommendations by assigning related assets in the flex asset or flex parent forms. (Typically, you configure relationships among flex parents, such as product parents, instead of individual assets.)

# Creating a Dynamic List Element

If your web site uses dynamic list recommendations, you must code the dynamic list elements that return lists of recommended assets. A dynamic list element is an instance of the CSElement asset type; this ensures that the dynamic list element will be transferred to the delivery system when the web site that uses it is published.

A dynamic list element must return a list named `AssetList`. The set of assets that becomes your `AssetList` must have the following traits:

- It must contain only assets of the types that you want to recommend.
- It must contain the IDs of the assets that you want to recommend.
- It should contain the assets' confidence ratings, although this is optional.

The following sample code is an excerpt from a dynamic list element:

```
1   <SEARCHSTATE.CREATE NAME="ssprod"/>
2   <SEARCHSTATE.ADDSIMPLESTANDARDCONSTRAINT NAME="ssprod"
    TYPENAME="PAttributes" ATTRIBUTE="BrowseCategory"
    VALUE="Fund Type"/>
```

Line 2 adds a constraint to the `ssprod` searchstate, filtering it to find items with a browse category of `Fund Type`.

```
3   <ASSETSET.SETSEARCHEDASSETS NAME="asprod"
    CONSTRAINT="ssprod" ASSETTYPES="Products"/>
```

Line 3 adds another constraint to the `ssprod` searchstate, creating an assetset composed entirely of `Product` assets.

```
4   <ASSETSET.GETASSETLIST NAME="asprod"
    LISTVARNAME="AssetList"/>
```

Finally, line 4 turns the assetset created in line 3 into the `AssetList` list.

When you have completed coding your dynamic list elements, provide their names and information about what sort of content they return to the users who will create the recommendation assets.

For information about creating recommendation assets, see the *Content Server Advanced Interface User's Guide*.

Chapter 41

# Coding Engage Pages

This chapter presents information about designing an online site that gathers visitor information and personalizes promotional messages for each visitor based on that information.

This chapter contains the following sections:


- Commerce Context and Visitor Context
- Identifying Visitors and Linking Sessions
- Collecting Visitor Data
- Templates and Recommendations
- Shopping Carts and Engage
- Debugging Site Pages


**Note**

This chapter refers to specific XML tags that you use to accomplish the tasks being described. In all cases, there are also equivalent JSP tags. The XML and JSP tags are all documented in the *Content Server Tag Reference*.

# Commerce Context and Visitor Context

During a visitor's session at an Engage site, a visitor context is created for that visitor. Five types of session objects are placed in the visitor context:

- Current shopping cart
- List of segments that the visitor belongs to
- List of promotions that the visitor qualifies for
- Time object that is used for calculating time-based rules for segments and promotions
- Utility object that gives you, the developer, access to product attributes

The commerce context encompasses the visitor context and gives you access to it.

There are two sets of XML and JSP object methods that serve as your interface to these contexts:

- Commerce context methods, which you use to place objects in the visitor context.
- Visitor Data Manager methods, which you use to gather, store, and retrieve visitor data and to associate a visitor's data with the correct visitor.

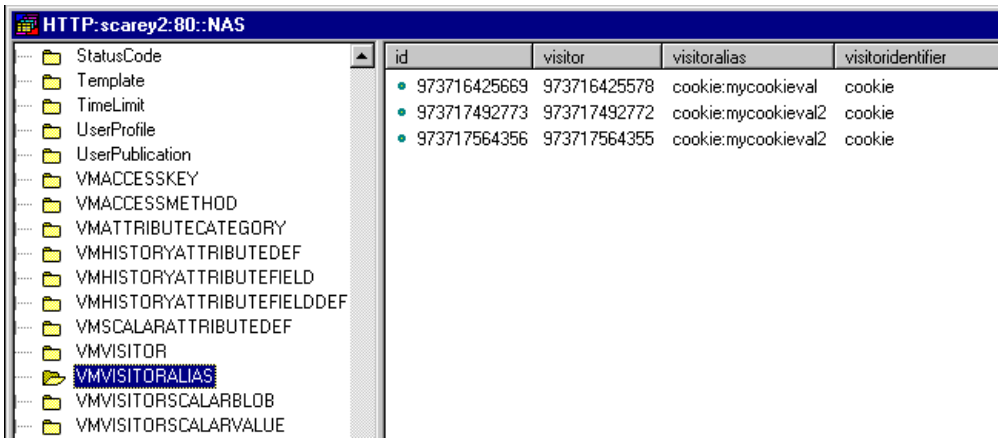
# Identifying Visitors and Linking Sessions

Engage creates a unique visitor ID for each visitor for each session. It stores these IDs in the `VMVISITOR` table in the Content Server database. The data gathered for a visitor during that session is identified by that visitor ID. To link the data gathered from one session to the data from another, your site pages must assign aliases that link those visitor IDs.

You use the following Visitor Data Manager object method to create an alias:

```
<VDM.SETALIAS KEY="keyvalue" VALUE="aliasvalue"/>
```

When you use this tag, Engage associates the visitor session ID with the alias, and writes them both to the `VMVISITORALIAS` table.




The values in this table link the data that is gathered in separate sessions to the same visitor because the alias provides a link to the visitor IDs that are recorded for that visitor. In the illustration above, the data recorded in the session associated with the visitor ID

973717492772 is linked to the data associated with the visitor ID 973717564355 because they have aliases with the same key/value pair.

All visitor information that is associated with sessions that are linked through common aliases — that is, aliases with the same key/value pairs — can be accessed during the current session. It is considered current visitor information.

You can create aliases with cookies, with login IDs, or with any other unique identifier that your site uses to recognize visitors.

The `VMVISITORALIAS` table grows quickly. For information about managing it and the other visitor data tables, see the *Content Server Administrator's Guide*.

# Collecting Visitor Data

To collect visitor data, you must program your online pages to gather it, validate it, and then write it to the Content Server database.

There are three Visitor Data Manager object methods that write this information to the database:

- `<VDM.SETSCALAR ATTRIBUTE="attribute" VALUE="value"/>` records visitor attributes.
- `<VDM.RECORDHISTORY ATTRIBUTE="attribute" LIST="valuelist"/>` records history definitions.
- `<VDM.SAVESCALAROBJECT ATTRIBUTE="attribute" OBJECT="objectname"/>` records visitor attributes of type binary. The demo site delivered with Engage uses this method to store shopping carts across sessions and to store saved searches for visitors.

**Note**

Because these tags write information to the database, they can be a factor in the performance of your delivery system. Be sure to use them efficiently.

These are the tables that store the visitor data:

| XML or JSP Object Method | Database Table That It Writes To |
|---|---|
| `VDM.SETSCALAR`<br>`vdm:setscalar` | `VMVISITORSCALARVALUE` |
| `VDM.SAVESCALAROBJECT`<br>`vdm:savescalarobject` | `VMVISITORSCALARBLOB` |
| `VDM.RECORDHISTORY`<br>`vdm:recordhistory` | `VMz----------`<br>(These tables are dynamically generated for each history definition. Engage creates a unique table for each one.) |

These tables grow quickly. For information about managing them, see the *Content Server Administrator's Guide*.

There are also a number of Visitor Data Manager object methods that retrieve this information from the Content Server database. See Chapter 10, "Error Logging and Debugging."

**Note**

For information about these and other Engage XML and JSP tags, see the *Content Server Tag Reference.*

## Coding Site Pages That Collect Visitor Data

This section presents an overview of the general steps that you follow when you code your site pages to collect visitor data:

1. Create forms to capture the data that you need your visitors to manually provide. It is a good practice to create form fields with names that match the names of the attributes that you created. (See Chapter 39, "Creating Visitor Data Assets," for more information.)

   Attributes are listed by their descriptions rather than by their names in the Engage Segment forms. Be sure that you do not confuse their attribute names with attribute descriptions when you are creating form fields or writing values to the Content Server database.

2. Create a "submit" page that validates the data that the visitor entered in the fields (either by using JavaScript or with a server-side validation method). The input data must comply with the constraints that you set for the attributes. For example, if you created a visitor attribute of type string with a length of 30, be sure that the form does not try to submit data from the form field with a length of 31.

3. Program the "submit" page to write the validated data to the Content Server database. Be sure to use the names of the attributes and history definitions and not their descriptions. Here are some examples:

### Example 1: Visitor Attributes

```
<!-- Write the registration information to the database.-->
<VDM.SETSCALAR ATTRIBUTE="name" VALUE="Variables.name"/>
<VDM.SETSCALAR ATTRIBUTE="age" VALUE="Variables.age"/>
<VDM.SETSCALAR ATTRIBUTE="jobdesc" VALUE="Variables.jobdesc"/>
```

### Example 2: History Definition

Because history definitions hold multiple values as an aggregate, you must create a list of the data before you can write it to the database. In this example, a form writes an order to the Content Server database:

```
<!-- Write the order details to a list. -->
<!-- assume that Variables.order_id is set to the order id -->
<!-- assume that Variables.wasCouponUsed is set to 1 (yes) or 0
(no) -->
<!-- assume that Variables.shippingtype is set to UPS or FedEx
-->
```

```
<!-- assume that Variables.order_price is set to the total
amount of the
order -->

<LISTOBJECT.CREATE NAME="histList" COLUMNS="orderid,
shippingtype, price, couponUsed"/>
<LISTOBJECT.ADDROW NAME="histList" orderid="Variables.order_id"
shippingtype="Variables.shippingtype"
price="Variables.order_price"
couponUsed="Variables.wasCouponUsed"/>
<LISTOBJECT.TOLIST NAME="histList" LISTVARNAME="itemList"/>

<!-- Write the list to the history definition named
visitorOrderHistory in the Content Server database.-->
<VDM.RECORDHISTORY ATTRIBUTE="visitorOrderHistory"
LIST="itemList"/>
```

And you can use that record to determine information about how many orders a visitor had made, when their first or last purchase was, and the total amount they've spent.

### Example 3: Visitor Attribute of Type Binary

Binary visitor attributes allow you to convert an object from the Content Server name space into a binary form. The sample site delivered with Engage uses two visitor attributes of type binary: one to store shopping carts across sessions and one to store saved searches.

To see these examples, use Content Server Explorer to examine `ElementCatalog/OpenMarket/Demos/CatalogCentre/GE/Navigation/stylesheet.xml` and `ElementCatalog/OpenMarket/Demos/CatalogCentre/GE/myge.xml`

4. If you want to gather data about visitor behavior (clickstream information, for example), you can program your pages to gather that without using input forms. For example, the demo site delivered with Engage uses a history definition to record the number of times a visitor browses the site.

   For this examples, use Content Server Explorer to examine `ElementCatalog/OpenMarket/Demos/CatalogCentre/GE/Navigation/stylesheet.xml`

5. Whenever visitor data is written to the database, segments and promotions can also change. Therefore, after any change to visitor data, be sure to recalculate the segments and promotions lists. There are two Commerce Context object methods that you can use:

   `COMMERCECONTEXT.CALCULATEPROMOTIONS`

   `COMMERCECONTEXT.CALCULATESEGMENTS`

   `COMMERCECONTEXT.CALCULATEPROMOTIONS` recalculates both the segments that the visitor belongs to and the promotions that apply to those segments.

6. Whenever visitor data is written to the database, ratings for assets can also change. Therefore, after any change to visitor data, be sure to refresh the ratings of any assets that are in an existing asset set.

   Use the `ASSETSET.ESTABLISHRATINGS` tag to refresh the asset ratings of the assets in a set.

**Note**

For information about these and other Engage XML and JSP object methods, see the *Content Server Tag Reference.*

# Templates and Recommendations

The key Commerce Context object method for invoking a recommendation asset is this one:

```
<COMMERCECONTEXT.GETRECOMMENDATIONS
COLLECTION="recommendationname" [LIST="inputlist" VALUE="rating"
MAXCOUNT="assetcount"] LISTVARNAME="assetlist"/>
```

This method retrieves and lists the assets that match the recommendation constraints passed to the method. It uses the following arguments:

- `COLLECTION` – the name of the recommendation. If you plan to use the same template for several recommendations, code the template to supply the identity of the recommendation through a variable.
- `LIST` – the name of the list of assets; this is the name that you want to be used as the input for the calculation.

  You use this argument when the recommendation named by `COLLECTION` is a context-based recommendation. Columns are `assettype` and `assetid`. You can create this list by creating a list object and adding rows for each asset that you want to use as input. For an example, use Content Server Explorer to examine `ElementCatalog/OpenMarket/Demos/CatalogCentre/GE/cart.xml`.
- `VALUE` – the default rating for assets that do not have one. If you do not declare a value, unrated assets are assigned a default rating of 50 on a scale of 0-100. FatWire recommends that you keep this value set to 50. For more information about the system default rating, see the recommendation asset section in the *Content Server Advanced Interface User's Guide*.
- `MAXCOUNT` – (optional) the maximum number of assets to return. Use this value to constrain the list of recommended assets.
- `LISTVARNAME` – the name that you want to assign to the list of assets. Its columns are: `assettype` and `assetid`.

If the segment list and the promotion list have not yet been created and placed in the visitor context, this object method invokes the methods that calculate them. Remember that promotions do not have templates—they override the template that a recommendation is using. If there are any promotions that apply to the current visitor **and** that override the recommendation named by the `COLLECTION` argument, the object method returns the ID of the promotion asset rather than the items identified by the recommendation asset.

**Note**

The `COMMERCECONTEXT.GETSINGLERECOMMENDATION` object method returns one recommended asset based on the recommendation criteria passed to the method. Typical uses for this method are to feature one product or to put one product on sale. See the *Content Server Tag Reference* for information about this object method and its JSP equivalent.

Before you begin coding the templates for recommendations, be sure to complete the following tasks:

- Meet with the marketing team to define all the merchandising messages that you want to display on your site and plan how to represent those messages in recommendations and promotions.

  For example, do you want to display a list of links to other products? What information should the link include? The product name only or also the price? What will be displayed when a recommendation returns a promotion rather than a list of assets?
- Determine where and on which pages the recommended assets from each recommendation will be displayed.

## Creating Templates for Recommendations

To use templates to render items that are returned by recommendation assets, you must complete at least the following basic steps:

1. Create a template element that invokes a recommendation asset. Use the object method described in the preceding section.
2. Code the template to display the items that are returned by the recommendation. The returned items are stored in a variable designated by the `LISTVARNAME` argument. This list includes the asset IDs and asset types of those items. Use that information to extract the asset attributes that you want to display. (Name, price, SKU, for example.)

   You can use the `ASSETSET.SETLISTEDASSETS` and `ASSETSET.GETASSETLIST` object methods to sort and display the returned assets and their attributes.

   For an example, use Content Server Explorer to examine `ElementCatalog/OpenMarket/Demos/CatalogCentre/Templates/GE/recommendation.xml`.
3. Open Engage. Under **New**, select **Template.** Create a corresponding Template asset for this template element. Enter a name that describes what the element does so that when you create a recommendation asset you know which template to assign to it. Identify the path to the element (its location in the ElementCatalog) in the **Element Name** field.
4. Publish the Template asset when other assets are published.
5. Render the recommendations on the appropriate site pages.

# Shopping Carts and Engage

The shopping cart interface is a CS-Direct Advantage feature. However, when you are using Engage, there are a number of additional facts and tips to keep in mind while you code your shopping cart pages:

- If your site uses promotions, you must code your cart pages to apply the discounts from the promotions.

  Use the `COMMERCECONTEXT.DISCOUNTCART` and `COMMERCECONTEXT.DISCOUNTTEMPCART` object methods to apply promotional discounts to the shopping cart.
- It is good practice to clear existing discounts from the cart before applying them again.
- You can store carts across sessions by writing them to the database as a visitor attribute of type binary (a scalar object). Be sure to write the cart object to the database each time the cart is modified.
- If your site uses a visitor login feature, there can be conditions under which you should merge shopping carts. For example, a visitor adds products to her cart before she logs in. Then, when she logs in, Engage finds a stored cart that also has items in it. In such a case, you would want to merge the carts.

For information about the `CART` object methods and their JSP equivalents, see the *Content Server Tag Reference*.

For an example of a Engage shopping cart, use Content Server Explorer to examine `ElementCatalog/OpenMarket/Demos/CatalogCentre/GE/cart.xml`.

# Debugging Site Pages

During your development phase, you must verify that session linking is set up correctly, that specific attributes obtain the value that you expect, and that recommendations return the items that you expect. There are several Engage object methods that you can use to retrieve and review information and values by writing information to a browser window or to the JRE log.

This section lists the Visitor Data Manager object methods that you will probably use the most. For information about these and any other XML and JSP object methods, see the *Content Server Tag Reference.*

## Session Links

Use the following Visitor Data Manager object methods to verify that pages that handle session linking are creating the aliases correctly:

- `<VDM.GETALIAS KEY="keyvalue" VARNAME="`*`varname`*`"/>`

  Retrieves an alias
- `<VDM.GETCOMMERCEID VARNAME="`*`varname`*`"/>`

  Retrieves the visitor's commerce ID from session data.
- `<VDM.GETACCESSID KEY="pluginname" VARNAME="`*`varname`*`"/>`

  Retrieves the visitor's access ID from session data.

## Visitor Data Collection

Use the following Visitor Data Manager object methods to retrieve values stored for specific visitor attributes, history attributes, and history definitions (records):

- `<VDM.GETSCALAR ATTRIBUTE="attribute" VARNAME="varname"/>`

  Retrieves a specific visitor attribute.

- `<VDM.LOADSCALAROBJECT ATTRIBUTE= "attribute" VARNAME= "varname"/>`

  Retrieves (materializes) an object stored as a visitor attribute of type binary.

- `<VDM.GETHISTORYCOUNT ATTRIBUTE="attribute" VARNAME="varname"[STARTDATE="date1" ENDDATE="date2" LIST="constraints"]/>`

  Retrieves the number of history definition records that were recorded for the visitor that match the specified criteria.

- `<VDM.GETHISTORYSUM ATTRIBUTE="attribute" VARNAME="varname" [STARTDATE="date1" ENDDATE="date2" LIST="constraints"] FIELD="fieldname"/>`

  Sums the entries in a specific field for the specified history definition.

- `<VDM.GETHISTORYEARLIEST VARNAME="varname" [STARTDATE="date1" ENDDATE="date2" LIST="constraints"]/>`

  Retrieves the timestamp of the first time the specified history definition was recorded for this visitor.

- `<VDM.GETHISTORYLATEST VARNAME="varname"[STARTDATE="date1" ENDDATE="date2" LIST="constraints"] />`

  Retrieves the timestamp of the last time (that is, the most recent time) the specified history definition was recorded for this visitor.

## Recommendations and Promotions

Use the following Commerce Context object methods to verify pages that display recommendations and promotions:

- `<COMMERCECONTEXT.CALCULATESEGMENTS/>`

  Lists the segments that the visitor belongs to. It examines the available visitor data, compares it to the data types that define the segments, and then lists the segments that are a match.

- `<COMMERCECONTEXT.GETPROMOTIONS LISTVARNAME="promotionlist"/>`

  Creates the list of promotions that the current visitor is eligible for

- `<COMMERCECONTEXT.GETRATINGS ASSETS="assetlist" LISTVARNAME="ratinglist" DEFAULTRATING="defaultrating"/>`

  Calculates the ratings of the assets in a named list according to how important the asset is to this visitor based on the segments that the visitor belongs to.

- `<COMMERCECONTEXT.GETSEGMENTS LISTVARNAME="segmentlist"/>`

  Retrieves the list of segments that the current visitor belongs to.

Chapter 42

# Memory-Centric Visitor Tracking

Site visitors who browse Engage assets typically provide information about themselves in a personal profile. Demographics information is also collected as visitors browse. To prevent overloading Content Server's database with large amounts of visitor data, and therefore to improve performance, memory-centric visitor tracking was developed for Engage assets.

This chapter describes the memory-centric method of tracking visitors and outlines requirements for implementing this method.

This chapter contains the following sections:


- Overview
- Enabling Memory-centric Visitor Tracking
- How Memory-centric Visitor Tracking Works

# Overview

When large numbers of visitors browse a web site, storing all of their personal information in a single repository degrades the performance of the delivery system. For this reason, sites that collect demographic information on visitors often require an additional repository to help improve site performance. Additional performance gain can be achieved by preventing load on Content Server's database.

As of version 7.5 patch 1, Content Server supports add-on repositories, enabling Engage developers to implement a repository of their own choice to store visitor scalar attribute values. Custom code must be written to store and retrieve visitor information to and from the repository. In addition, Content Server's memory-centric visitor tracking method must be enabled. Differences between memory- and database-centric methods are outlined in this overview. Information about enabling memory-centric tracking is provided in the rest of this chapter, followed by diagrams illustrating how memory-centric visitor tracking works.

## Database-centric Model

In database-centric visitor tracking, performance issues on Engage-enabled web sites arise for the following reasons:

- All visitor information is stored in Content Server's database. Multiple database accesses are required to store and retrieve information.
- When dynamic recommendations are used and a large amount of data is returned, Content Server logs a greater number of `SystemItemCache` entries, which leads to performance degradation. To restore performance, the `render:overridedeps` tag was introduced as a way of limiting the number of dependencies that are logged in Content Server's database. This tag applies to both the database- and memory-centric models.

## Memory-centric Model

Memory-centric tracking improves system performance by reducing load on Content Server's database.

- All computations are performed in memory:
  - When custom code retrieves visitor scalar attribute values (such as age and gender) from the add-on repository, the memory-centric model stores the retrieved values in memory and uses them with history attribute values to compute segments to which the visitor belongs.
  - Memory-centric tracking computes statistics on various history attribute values in memory and then caches the statistics. Statistics include sums, counts, oldest, and newest.
- Alias and history attribute values are stored in Content Server's database, but in a way that reduces both the volume of visitor information and the number of calls required to access the information.

As in database-centric tracking, developers can use the `render:overridedeps` tag to specify the number and types of Engage asset dependencies to log in Content Server's database. Enabling memory-centric tracking requires setting a property in `visitor.ini` and writing supporting template code.

# Enabling Memory-centric Visitor Tracking


- Visitor Tracking Property
- Supporting Code
- Batch-Saving History Attributes to the Database


## Visitor Tracking Property

Memory-centric visitor tracking is enabled by the property `vis.useSessionVisitorConnection`, which must be manually added to Content Server's `visitor.ini` file **on the delivery system**:

- Setting `vis.useSessionVisitorConnection=true` enables memory-centric visitor tracking. Supporting template code must also exist on the delivery system in order for Engage visitors to be correctly tracked; pre-existing code can be reused. For more information, see "Supporting Code."
- Setting `vis.useSessionVisitorConnection=false`, leaving it blank, or omitting it from `visitor.ini` enables the database-centric method.

**Note**

On all content management systems, `vis.useSessionVisitorConnection` must be either set to `false` or omitted, thus enabling the database-centric method, which allows visitor attributes to be created and otherwise managed. (Visitor attribute management is supported only on content management systems that are enabled for database-centric tracking. It is not supported on delivery systems; visitor attributes must be published to the delivery system.)

## Supporting Code

The skeleton template in this section shows the type of code that must be written to support memory-centric tracking.

**Note**

How the template code executes depends on the value of the `vis.useSessionVisitorConnection` property. If the value is set to `false`, the database method of tracking visitors is used. If the value is set to `true`, memory-centric tracking takes effect.

```
vdm:setalias
retrieve visitor scalar attributes from add-on repository
vdm:setscalar
commercecontext:calculatesegments
commercecontext:getrecommendations
render Engage assets
vdm:recordhistory
...
render:overridedeps
```

For more information about this template, see "How Memory-centric Visitor Tracking Works."

## Batch-Saving History Attributes to the Database

As of Content Server 7.5 Patch 2, history attributes are saved first to memory, then to the file system, and finally as a batch to Content Server's database. Batch-saving prevents excessive memory usage (which would otherwise occur when data is produced faster than it can be saved to the database). The batch-save process batches one `HistoryAttributeDef` table at a time from the file and saves the tables to the database in bulk.

Batch saving to the database requires Engage to have access to database information. Database access is enabled by adding a file named `<dataSourceName>.properties` to the classpath to specify the following information: `driver`, `url`, `user`, and `password`. For example, if the data source name is `csDataSource`, then a file named `csDataSource.properties` must be placed in the classpath. In our example, properties in the file are set as follows:

```
driver=com.jnetdirect.jsql.JSQLDriver
url=jdbc:JSQLConnect://localhost:1433/database=TomcatDB
user=tomcatuser
password=tomcatuser
```

The following Java JVM parameter was added to control the time interval at which threads are spawned for asynchronous batch-saving of history attributes from memory to Content Server's database via the file system:

```
-Dvisitor.SyncInterval=<seconds>
```

If left unspecified, the parameter value defaults to 30 seconds.

# How Memory-centric Visitor Tracking Works

Diagrams in the rest of this chapter show how memory-centric tracking works.

**Note**

In the diagrams that follow, the add-on repository is a system of your own choice, used to store and retrieve visitor scalar attribute values. Custom code must be written to store and retrieve the values.

# 1. Visitor Detection

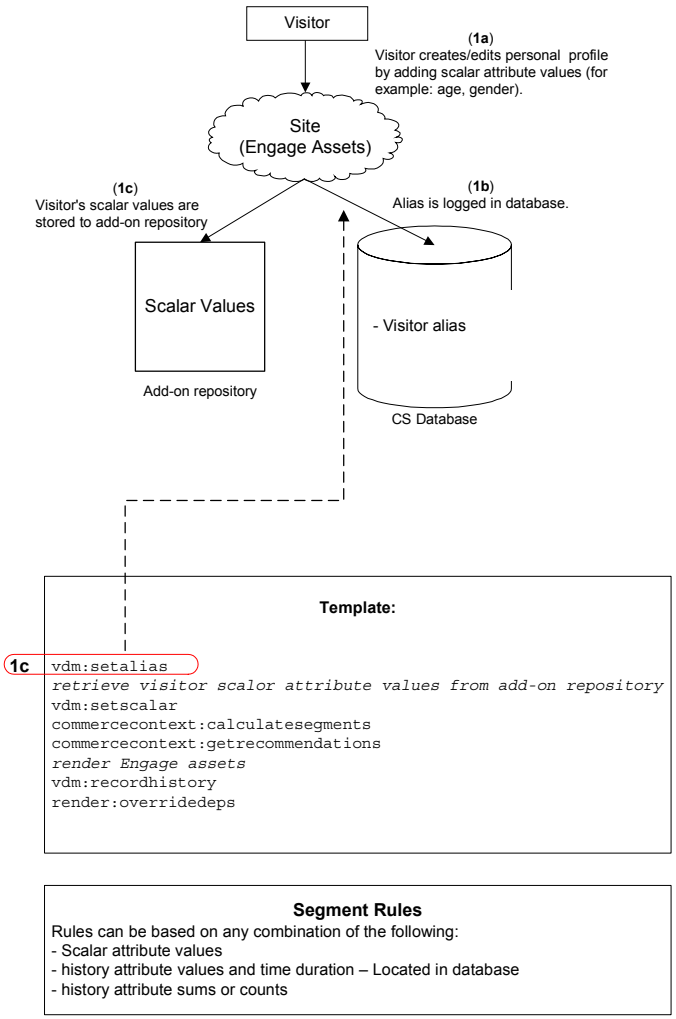
**1a.** A first-time visitor enters the site and creates a personal profile. The following events occur:

**1b.** The `vdm:setalias` tag in the template code is used to log the visitor's alias to Content Server's database. In subsequent sessions, the visitor is automatically recognized. As a result, only a single ID unique to the visitor is kept in the database, which improves performance.

Note that memory-centric visitor tracking logs a *unique ID per visitor*, whereas the existing database-centric method logs a *unique ID per visit* — even for returning visitors. Extra IDs create extra load on the database and reduce performance.

**1c.** The visitor's scalar values (personal information, such as age and gender) are stored to the add-on repository (by custom code).

In the remaining steps, Content Server gathers and computes information that it needs to determine the visitor's segments and therefore which Engage assets to display to the visitor.



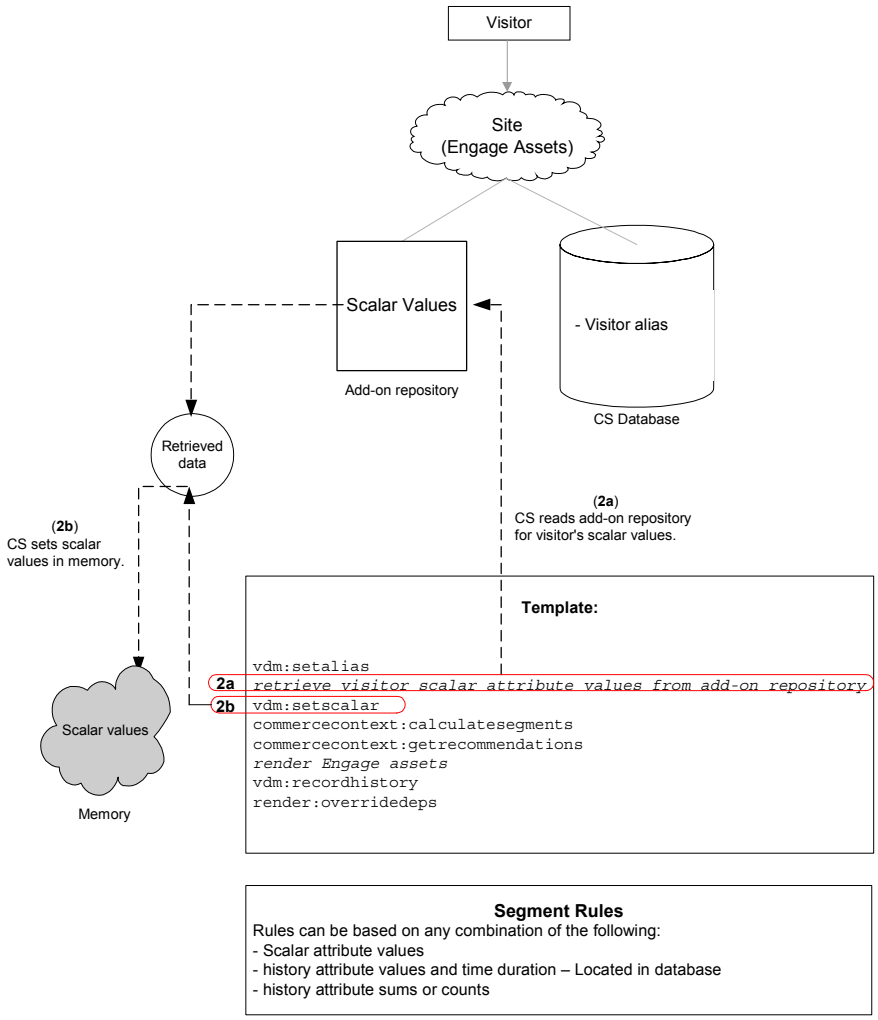
## 2. Retrieving Scalar Values

Content Server begins computing the visitor's segments. Starting with scalar values:

**2a.** Content Server retrieves scalar values from the add-on repository (using custom code).

**2b.** Content Server sets retrieved scalar values in memory, using the `vdm:setscalar` tag.

The session-based implementation stores visitor information (scalar values) only in the http session. Therefore, when a new session begins, visitor information is made available through the add-on repository, instead of Content Server's database. A new level of caching has been added to optimize the performance of querying for visitor scalar values.



## 3. Collecting History Attribute Values

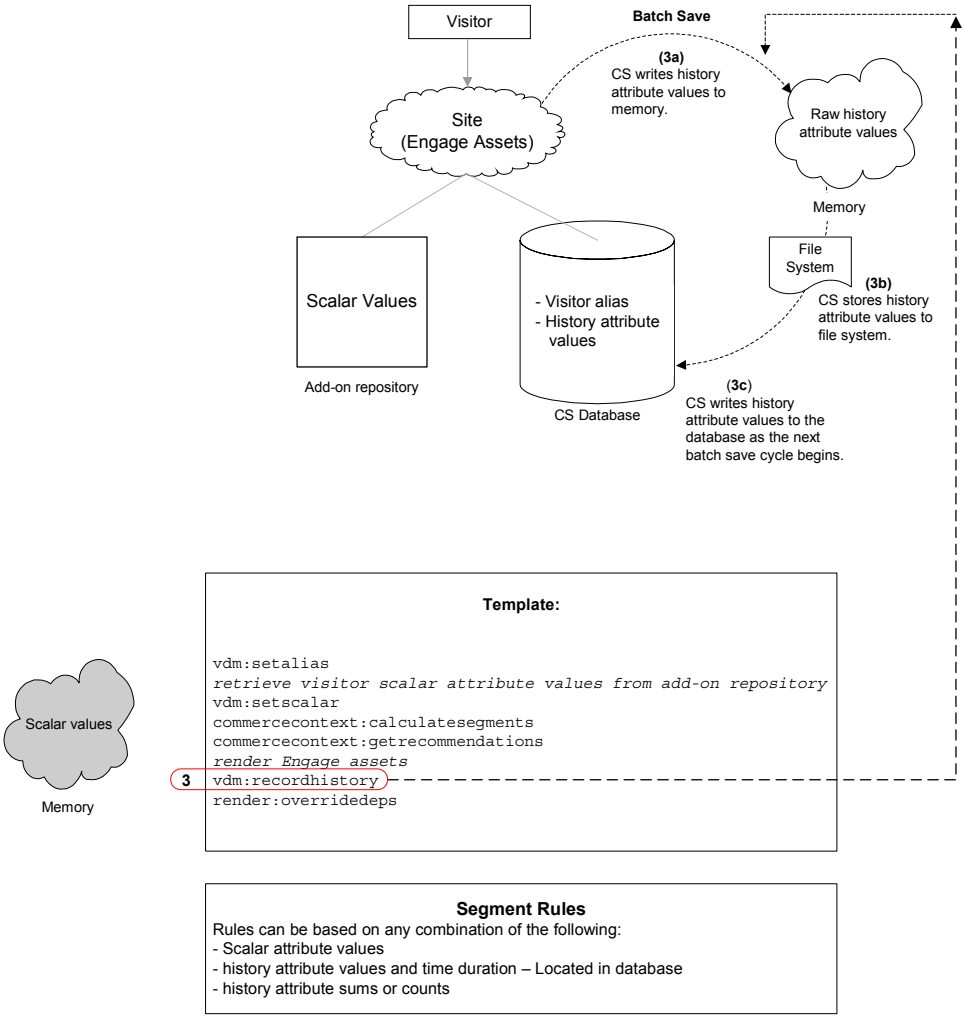
History attribute values are also required for computing a visitor's segments. During the session, history attribute values are collected cyclically:

**3a.** The `vdm:recordhistory` tag is used to collect history attribute values into memory at one-minute intervals.

**3b.** The history attribute values are saved to the file system and then written to Content Server's database in batches. The memory is cleared and the cycle starts again.

Visitor history attribute values are collected first in memory to avoid the need for acquiring database connections at each call. They are saved to the file system to enable batch saving to the database, which minimizes memory and database connections usage, and increases the availability of history attribute values.

When history attribute values are committed to the database, computation of sums and counts begins, as shown on page 810.



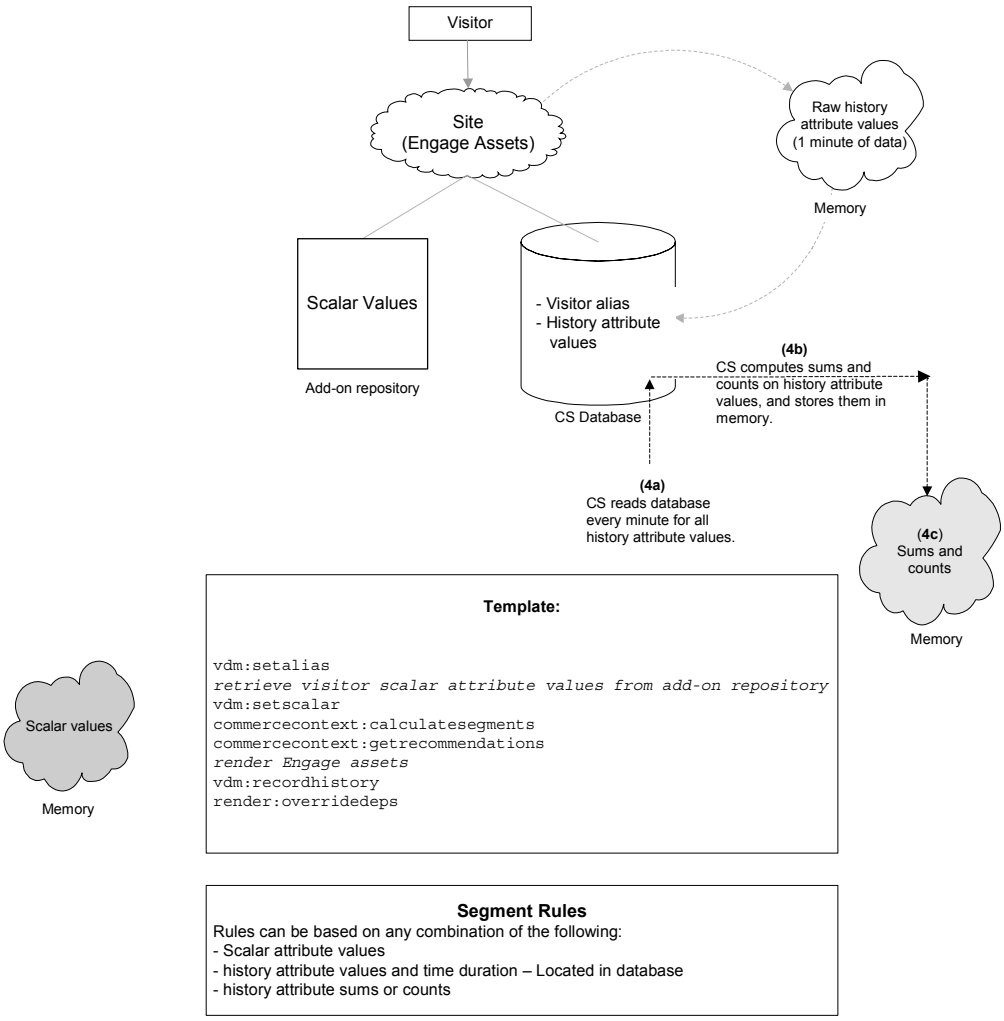
## 4. Computing Sums and Counts

After the first set of history attribute values is collected and written to the database (step 3), Content Server:

**4a.** reads the database at 1-minute intervals, computes the sums and counts, and

**4b.** stores the results in memory.

Caching of sums and counts significantly reduces database queries, while the 1-minute interval makes the computation of sums and counts accurate to within 1 minute.



## 5. Computing Segments

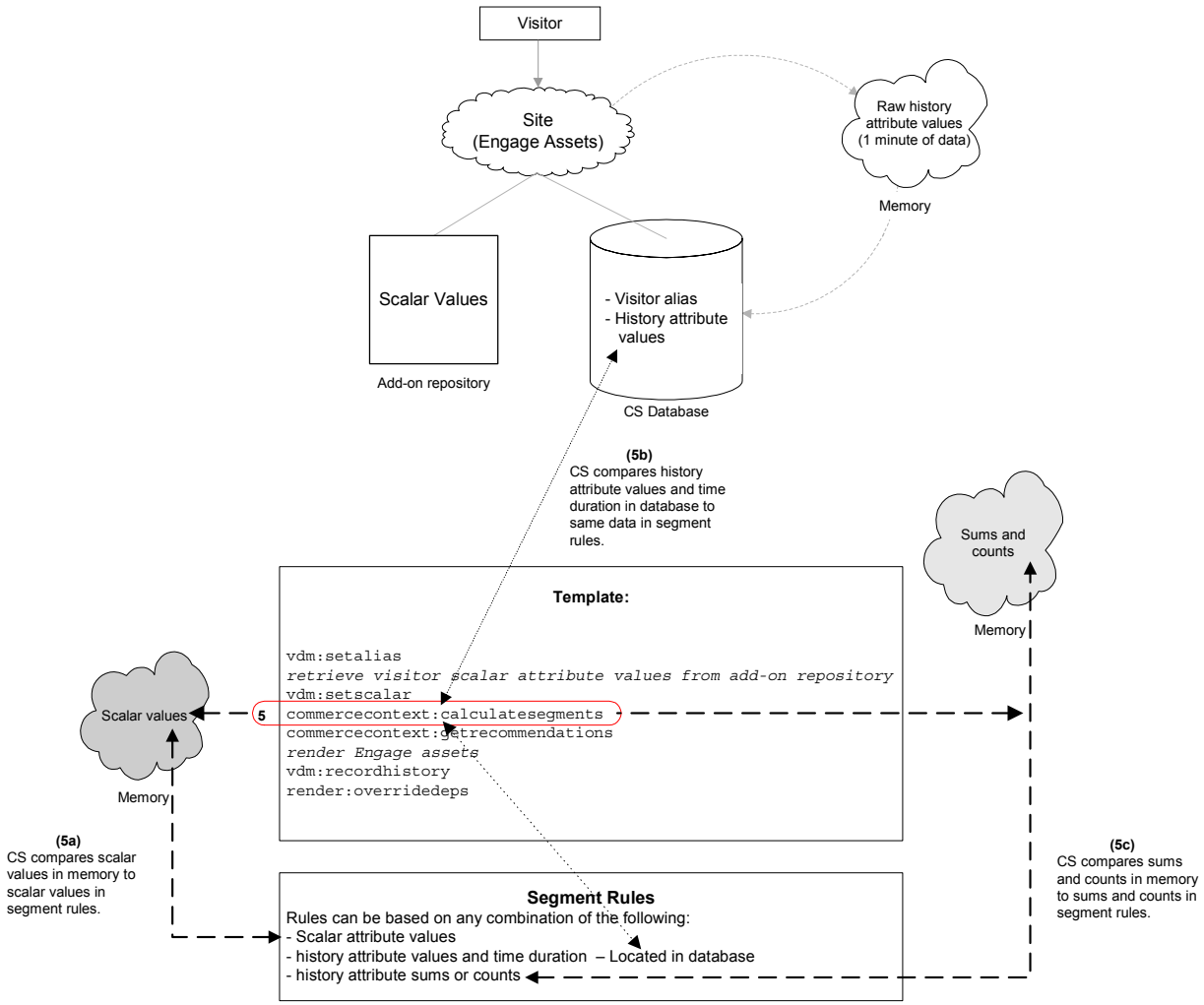
At this point, Content Server has enough information to compute the visitor's segments. Using the `commercecontext:calculatesegments` tag, Content Server does the following:

**5a.** Compares the visitor's scalar values in memory to those in the segment rules.

**5b.** Compares the history attribute values in the database to those in the segment rules.

**5c.** Compares the sums and counts in memory to the sums or counts in the segment rules

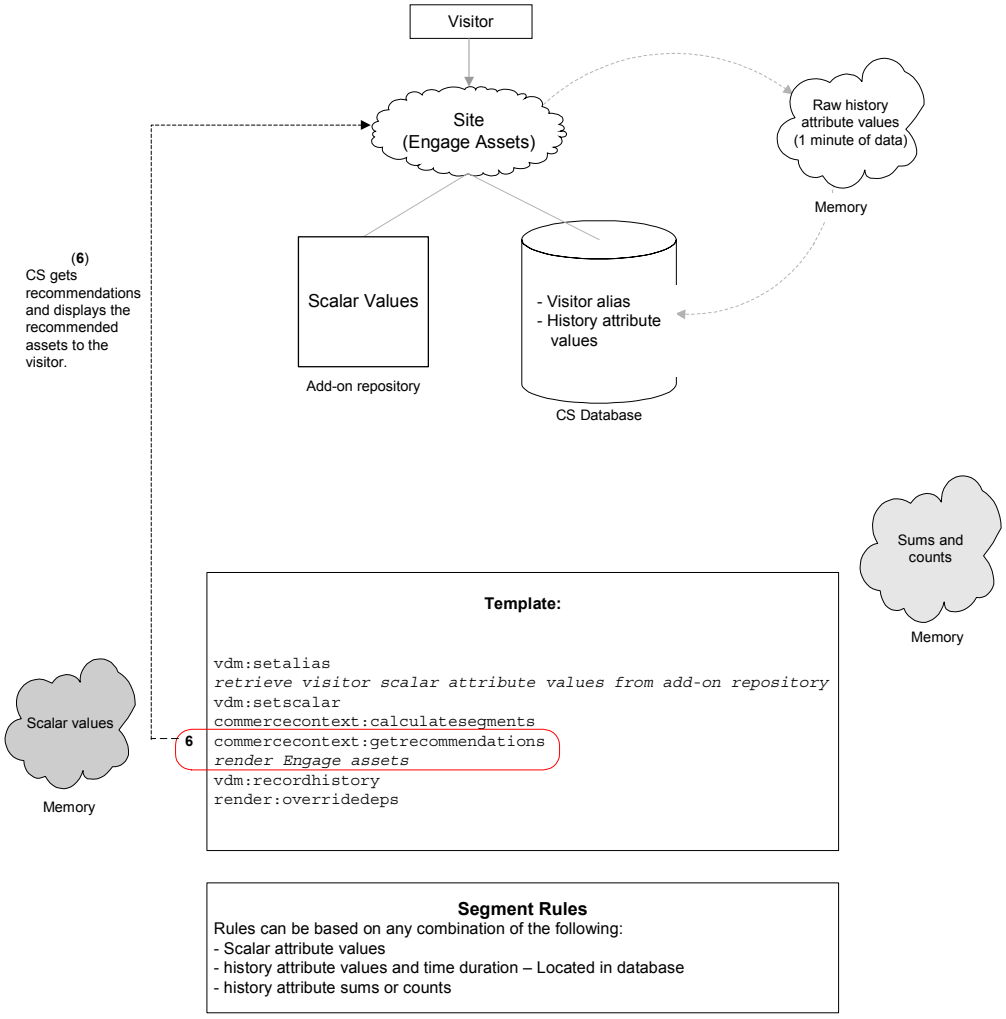
From the comparisons, Content Server determines the visitor's segments.



## 6. Determining and Displaying Recommended Assets

Having determined the visitor's segments, Content Server uses the `commercecontext:getrecommendations` tag to determine which Engage assets are of most interest to the visitor. Templates for Engage assets display the recommended assets to the visitor.

Further performance gains are achieved by use of the `render:overridedeps` tag, as explained on page 813.



## Logging Dependencies

The `render:overridedeps` tag is used to reduce the number of `SystemItemCache` entries logged. The tag must be inserted at the end of the template (or CSElement), just before the `</cs:ftcs>` tag, to remove all the existing dependencies logged on the page and to log the dependencies that are specified by the `render:overridedeps` tag. The possible dependencies are:

- Unknown dependencies
- Dependencies on a single specific asset
- Unknown dependencies for a specific asset type

The `render:overridedeps` tag takes the following parameters:

- `cid`: ID of the asset of type `c`. When combined with `c`, it works like asset, a dependency will be logged against the asset determined by `c` and `cid`.
- `c`: Type of asset.
- `deptype`: Dependency type.

Which parameters are used and how they are set determines the dependency that is logged:

- When `deptype` alone is specified and set to `unknowndeps`, `render:overridedeps` logs an unknown dependency.
- When `c` and `cid` are both specified, `render:overridedeps` logs a dependency on the asset specified by `c` and `cid`.
- When `c` and `deptype='unknowndeps'` are both specified, `render:overridedeps` logs an unknown dependency on the asset type specified by `c`.

**Caution!**

Use `render:overridedeps` **carefully**. The only way to flush page caches is to edit the asset for which dependencies are logged. For example, if you are logging dependencies for a single asset (or asset type), you can flush the cache by editing the asset for which dependencies are logged. Users must know which asset(s) to edit.

Part 8

# API Reference

This part describes how to program your online site to deliver the data (assets) that you have designed.

It contains the following chapters:


- Chapter 43, “Asset API”
- Chapter 44, “Public Site Search”

Chapter 43

# Asset API

The purpose of this tutorial is to introduce the Asset API to the developer. It is meant to be a quick reference and not a substitute for *Javadocs*. Code samples throughout this tutorial illustrate how the Asset API is used.

This chapter contains the following sections:


- Overview
- Primary Interfaces
- Getting Started
- Asset API Read
- Asset API Write
- Development Strategies
- Optional: Setting Up to Use the Asset API from Standalone Java Programs

# Overview

The Asset API is a Java API used to access and manipulate Content Server assets. Its main purpose is to broaden the context in which assets can be handled. Its major features are the following:

- The Asset API supports Content Server in a non-servlet context.

  Prior to the Asset API, Content Server exposed primary interfaces for programming in the form of tags (both in XML and JSP), used as the means for creating web pages and applications. In this sense, Content Server was tightly built around the servlet model and was not usable in other contexts, such as standalone Java programs (EJB, for example). Therefore, there was a need to create an API that could be used anywhere, regardless of the servlet framework.
- The Asset API unifies the retrieval, creation, and modification of the two asset families: basic and flex
  - The Asset API represents both asset families with `AssetData` and `AttributeData`
  - The Asset API uses generic `Condition` and `Query` objects for both flex and basic assets
- The Asset API supports the creation and editing of basic assets, flex assets, and flex parent assets in the form of Java objects.

**Note**

The Asset API does not log any dependencies other than the asset that is being loaded.

# Primary Interfaces

The Asset API provides access to data as well as definitions for Content Server assets:

- Package `com.fatwire.assetapi.data` contains classes that are useful in reading data.
- Classes under `com.fatwire.assetapi.def` are for asset definitions.
- Package `com.fatwire.assetapi.query` contains constructs necessary for building a Query. For further details, see the *Javadoc*.

The Asset API defines the following primary interfaces:

- **`Session`:** The primary entry point into Content Server from the API. One has to obtain a session to be able to do anything at all in the Asset API.
- **`AssetDataManager`:** A manager for reading asset data. Developers can query for information here, as well as look up asset associations and other information.
- **`AssetTypeDefManager`:** A manager for reading an asset type's definition. 'Definition' is a loaded term in Content Server where flex assets are concerned. Here it is used in the generic sense—as something that defines the structure of an asset type. As a result, basic asset types also have a definition.

- **`AssetData`:** An asset's data; basically a collection of `AttributeData` instances and other information about the asset itself.
- **`AssetId`:** The asset type-id combination.
- **`DimensionableAssetManager`**: A manager that supports multilingual assets by retrieving translations of any given asset.

# Getting Started

- FirstSite II (FSII) is required to run the examples in this tutorial. The `tools.jar` file (available in JDK 1.5 and above) must be in the classpath.

> **Note**
>
> The Asset API can be used from a standalone Java program. Doing so requires some configuration. For details, see "Optional: Setting Up to Use the Asset API from Standalone Java Programs," on page 837.

- Working through the examples in this chapter requires a knowledge of `jsp` elements and how they are created in the Content Server environment. For information and procedures, see the chapter named "Creating Template, CSElement, and SiteEntry Assets" in the *Content Server Developer's Guide*.

# Asset API Read

With this brief background, let's start writing some code based on FSII data. Here are the topics we'll be covering in this section:

- A Simple Example: Reading Field Values
- Reading AssetId
- Reading Attributes Given the Asset ID
- Query
- Complex Query
- Sorting
- Reading BlobObject
- Retrieving Multi-Valued Attributes
- Multilingual Assets: Retrieving Translations
- Reading Asset and Attribute Definitions
- Reading Key-Value Mappings

## A Simple Example: Reading Field Values

Let's try to read all the values of the "FSIIHeadline" field in FSII Articles. Here are the steps to follow (they are explained below the sample code, on page 821):

1. Get a session.

2. Get a handle to `AssetDataManager`.
3. Build a query.
4. Perform 'read' and print the results.

Here is the code that implements these steps (in a `jsp` element):

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<%@ page import="java.util.*"%>
<cs:ftcs>
<%
Session ses = SessionFactory.getSession();
AssetDataManager mgr =
              (AssetDataManager) ses.getManager(
              AssetDataManager.class.getName() );
Query q = new SimpleQuery("Content_C", "FSII Article", null,
              Collections.singletonList("FSIIHeadline") );
for( AssetData data : mgr.read( q ) )
{
    out.println( data.getAttributeData("FSIIHeadline").getData() );
    out.println( "<br/>" );
}
%>
</cs:ftcs>
```

1. `SessionFactory.newSession()` builds a session for a given user. From that point on, all data that is read using this session instance is based on the user's ACL permissions. You could also simply call `newSession( null, null )` and get a Session that belongs to `DefaultReader`, an assumed user. CS-powered web applications generally run as this user at runtime. However, if the user name and password are specified and happen to be incorrect, you will get an exception.
2. Using the session, get a handle to `AssetDataManager.getManager`. (The `AssetDataManager.class.getName()` method does this.)
3. A `Query` represents results that are based on the user's search criteria. In this example, we are using a simple version of `Query`, where we specify the asset type (`Content_C`) subtype (`FSII Article`) and the list of attributes to be returned (just `FSIIHeadline` in this case). We want all assets; therefore the third parameter (which takes `Condition` instance) is `null`. We will see how to use `Conditions` later on (in "Complex Query," on page 824).
4. The `read()` method of `AssetDataManager` returns an `Iterable` over `AssetData`. Each piece of asset data contains an instance of `AttributeData` against an attribute name. `AttributeData.getData()` returns the real data of the attribute itself.

## Reading AssetId

What if we wanted to know the id's of all these assets? `AssetData.getAssetId()` returns an `AssetId` instance.

The above code can be modified like this to print `AssetId`:

```
for( AssetData data : mgr.read( q ) )
{
AssetId id = data.getAssetId();
out.println( data.getAttributeData("FSIIHeadline").getData() + "
             id=" + id );
out.println( "<br/>" );
}
```

The code above prints the following lines (note that `AssetId` is a composite—it contains the `id` number as well as type. (`AssetId.getId()` and `AssetId.getType()` return `id` and type separately):

```
AudioCo. America Announces H300 series id=Content_C:1114083739888
AudioCo. New Portable Media Player Offers Full Video Experience
   id=Content_C:1114083739926
AudioCo.'s First Under Water MP3 Player id=Content_C:1114083739951
...
```

## Reading Attributes Given the Asset ID

What if we already have an asset `id`, by some means? (either passed into a template or acquired programmatically). How can we read attributes of that? Let's consider an `AssetId (Content_C:1114083739888`, for example, as in the highlighted code above) and attempt to print its name, description, and FSIIBody. The following code does this:

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.openmarket.xcelerate.asset.*"%>
<cs:ftcs>
<%
    Session ses = SessionFactory.getSession();
    AssetDataManager mgr =
              (AssetDataManager) ses.getManager(
              AssetDataManager.class.getName() );
    AssetId id = new AssetIdImpl( "Content_C", 1114083739888L );
    List attrNames = new ArrayList();
    attrNames.add( "name" );
    attrNames.add( "description" );
    attrNames.add( "FSIIBody" );

    AssetData data = mgr.readAttributes( id, attrNames );
    AttributeData attrDataName = data.getAttributeData( "name" );
    AttributeData attrDataDescr = data.getAttributeData( "description"
              );
    AttributeData attrDataBody = data.getAttributeData("FSIIBody");

    out.println( "name:" + attrDataName.getData() );
    out.println( "<br/>" );
    out.println( "description:" + attrDataDescr.getData() );
    out.println( "<br/>" );
    out.println( "FSII Body:" + attrDataBody.getData() );
```

```
    out.println( "<br/>" );
%>
</cs:ftcs>
```

Here, we are indicating which attributes to read for a given `id`. As you can see, you can specify basic fields (such as name, description) as well as flex attributes; both are treated as attributes.

Alternatively, you can load all attributes for a given `id` by using `AssetDataManager.read( List<AssetId> ids )`. The following code demonstrates how this is done for a single `AssetId`:

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.openmarket.xcelerate.asset.*"%>
<cs:ftcs>
<%
    Session ses = SessionFactory.getSession();
    AssetDataManager mgr =
              (AssetDataManager) ses.getManager(
              AssetDataManager.class.getName() );
    AssetId id = new AssetIdImpl( "Content_C", 1114083739888L );

    Iterable<AssetData> dataItr = mgr.read( Collections.singletonList(
              id ) );

    for( AssetData data : dataItr )
    {
      for(AttributeData atrData : data.getAttributeData() )
      {
         out.println( "<br/>" );
         out.println( "attribute name:" + atrData.getAttributeName()
                 );
         out.println( "data: " + atrData.getData() );
      }
    }
%>
</cs:ftcs>
```

## Query

A `Query` specifies the criteria based on which data is looked up. What if we want to look up a product whose SKU is `iAC-008`. The code below does this and prints the name and `id`:

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<cs:ftcs>
<%
Session ses = SessionFactory.getSession();
```

```
      AssetDataManager mgr = (AssetDataManager) ses.getManager(
              AssetDataManager.class.getName() );
Condition c = ConditionFactory.createCondition( "FSIISKU",
              OpTypeEnum.EQUALS, "iAC-008" );
Query query = new SimpleQuery( "Product_C", "FSII Product", c,
              Collections.singletonList( "name" ) );

for( AssetData data : mgr.read( query ) )
{
    AttributeData attrData = data.getAttributeData( "name" );
    out.println( "name:" + attrData.getData() );
      out.println( "<br/>" );
    out.println( "id:" + data.getAssetId() );
}
%>
</cs:ftcs>
```

`Query` consists of a `Condition`, set of Attributes to be returned, and a `SortOrder`. The example above uses a condition that is built on the `FSIISKU` attribute value being `EQUAL` to `iAC-008`. Just as we did in the previous example, we pass in the list of attribute names we want to be returned in the resulting collection of `AssetData`.

There are some considerations as to what types of attributes can be queried and how. (See "Query Types," on page 834 for a complete discussion of different query algorithms.) In short, some types of queries are possible with one algorithm, but not with the other. Note that this is exactly the behavior we have in the `Asset` family of tags and `AssetSet` family of tags. To illustrate this point, say we want to read all products whose price (`FSIIPrice`) is greater than 179.

`FSIIPrice` is of type `MONEY`. Consulting Table 18 in ("Query Types," on page 834), we see that the `GREATER_THAN` operation is allowed for this data type only in the basic/generic algorithm. The following code uses that algorithm to get all products whose price is greater than 179. The choice of query algorithm is made by the highlighted line in the code below:

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<cs:ftcs>
<%
    Session ses = SessionFactory.getSession();
        AssetDataManager mgr = (AssetDataManager) ses.getManager(
              AssetDataManager.class.getName() );
    Condition c = ConditionFactory.createCondition( "FSIIPrice",
              OpTypeEnum.GREATER_THAN, 179.0f );
    Query query = new SimpleQuery( "Product_C", "FSII Product", c,
              Arrays.asList( "name", "FSIIPrice" ) );
    query.getProperties().setIsBasicSearch( true );

    for( AssetData data : mgr.read( query ) )
    {
      AttributeData name = data.getAttributeData( "name" );
      AttributeData price = data.getAttributeData( "FSIIPrice" );
```

```
        out.println( "name:" + name.getData() );
        out.println( "id:" + data.getAssetId() );
        out.println( "price:" + price.getData() );

        out.println( "<br/>" );
      }
%>
</cs:ftcs>
```

## Complex Query

A complex query can be achieved through nested `Conditions`. According to the choice of query algorithm, we are subjected to the constraints listed in "Query Types," on page 834.

The following code retrieves all the `Product_C` assets whose `FSIIPrice` attribute is greater than `179.0` and/or whose names are like "`FSII`":

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<cs:ftcs>
<%
Session ses = SessionFactory.getSession();
     AssetDataManager mgr = (AssetDataManager) ses.getManager(
             AssetDataManager.class.getName() );
Condition c1 = ConditionFactory.createCondition( "FSIIPrice",
             OpTypeEnum.GREATER_THAN, 179.0f );
Condition c2 = ConditionFactory.createCondition( "name",
             OpTypeEnum.LIKE, "FSII" );
Condition c = c1.and( c2 ); // c1.or( c2 );

Query query = new SimpleQuery( "Product_C", "FSII Product", c,
             Arrays.asList( "name", "FSIIPrice" ) );
query.getProperties().setIsBasicSearch( true );

for( AssetData data : mgr.read( query ) )
{
    AttributeData name = data.getAttributeData( "name" );
    AttributeData price = data.getAttributeData( "FSIIPrice" );

    out.println( "name:" + name.getData() );
    out.println( "id:" + data.getAssetId() );
    out.println( "price:" + price.getData() );

    out.println( "<br/>" );
}
%>
</cs:ftcs>
```

## Sorting

What if we wanted to retrieve the results sorted by price? The following code does that, by specifying a `SortOrder`, ascending in this example. To reverse the sort order, change `true` to `false` in the highlighted line of the code below:

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<cs:ftcs>
<%
Session ses = SessionFactory.getSession();
        AssetDataManager mgr = (AssetDataManager) ses.getManager(
             AssetDataManager.class.getName() );

SortOrder so = new SortOrder( "FSIIPrice", true );
Query query = new SimpleQuery( "Product_C", "FSII Product",
     null, Collections.singletonList( "FSIIPrice" ),
             Collections.singletonList( so ) );

for( AssetData data : mgr.read( query ) )
{
    AttributeData price = data.getAttributeData( "FSIIPrice" );

    out.println( "id:" + data.getAssetId() );
    out.println( "price:" + price.getData() );

    out.println( "<br/>" );
}
%>
</cs:ftcs>

The above code sorts and prints asset ids and price in the ascending
             order of FSIIPrice,

id:Product_C:1114083739851 price:89.99
             id:Product_C:1114083739757 price:99.95
             id:Product_C:1114083739696 price:129.99
             id:Product_C:1114083739301 price:129.99
             id:Product_C:1114083739471 price:179.95
             id:Product_C:1114083739350 price:189.95
             id:Product_C:1114083739225 price:399.99
             id:Product_C:1114083739596 price:899.95
             id:Product_C:1114083739804 price:1399.99
             id:Product_C:1114083739549 price:3799.95
             id:Product_C:1114083739663 price:6999.99
```

## Reading BlobObject

The Asset API defines a special class to represent file type of data, data that is stored as a binary file. For example, the `FSIIDocumentFile` attribute in FirstSite II is of type blob.

The following code reads `FSIIDocumentFile` from all `Document_C` instances and prints the asset `id`, file name, and file size.

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<cs:ftcs>
<%
Session ses = SessionFactory.getSession();
       AssetDataManager mgr = (AssetDataManager) ses.getManager(
              AssetDataManager.class.getName() );

Query query = new SimpleQuery( "Document_C", "FSII Document", null,
              Collections.singletonList( "FSIIDocumentFile" ) );

for( AssetData data : mgr.read( query ) )
{
    AttributeData docAttr = data.getAttributeData("FSIIDocumentFile");
    BlobObject fileObj = (BlobObject)docAttr.getData();
    byte [] d = new byte[fileObj.getBinaryStream().available()];
    fileObj.getBinaryStream().read(d);

    out.println( "id:" + data.getAssetId() );
    out.println( "file name:" + fileObj.getFilename() );
    out.println( "file size:" + d.length );

    out.println( "<br/>" );
}
%>
</cs:ftcs>
```

## Retrieving Multi-Valued Attributes

The Asset API supports multi-valued attributes in the same way it supports single-valued attributes. `AttributeData` contains a companion method, `getDataAsList()` to retrieve multiple values. The following code attempts to print data contained in `FSIIKeyword`, a multi-valued attribute.

**Note**

Because sample data that ships with FirstSite II does not have data for the `FSIIKeyword` attribute, this sample code does not print any keywords. Before running this code, edit some of the `FSIIDocument` instances to add data for this attribute.

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<cs:ftcs>
<%
```

```
Session ses = SessionFactory.getSession();
       AssetDataManager mgr = (AssetDataManager) ses.getManager(
              AssetDataManager.class.getName() );

Query query = new SimpleQuery( "Document_C", "FSII Document",
     null, Collections.singletonList( "FSIIKeyword" ) );

for( AssetData data : mgr.read( query ) )
{
    AttributeData attrData = data.getAttributeData( "FSIIKeyword" );
    List retData = attrData.getDataAsList();
    out.println( "id:" + data.getAssetId() );

    for( Object o : retData )
    {
      out.println( "data:" + o );
    }

    out.println( "<br/>" );
}
%>
</cs:ftcs>
```

## Multilingual Assets: Retrieving Translations

The Asset API also provides interfaces and methods to deal with multilingual assets. Basically, you have methods in `DimensionableAssetManager` to handle multilingual assets. They deal with getting all the locales for a given asset and specific translation of an asset.

The following example first retrieves the translations for asset `Page: 1118867611403` by using the `getRelatives` method in `DimensionableAssetManager` with group set to `Locale`, and then it uses the `getRelative` method to get the `fr_FR` translation of the asset.

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="com.fatwire.mda.DimensionableAssetManager"%>
<%@ page import="com.openmarket.xcelerate.asset.*"%>
<%@ page import="java.util.*"%>

<cs:ftcs>
<%
Session ses = SessionFactory.getSession();
DimensionableAssetManager mgr =
  (DimensionableAssetManager)ses.getManager(DimensionableAssetManager.
  class.getName());

AssetId page_asset = new AssetIdImpl("Page", 1118867611403L);

for( AssetId id : mgr.getRelatives( page_asset, null, "Locale" ))
{
       out.println( id );
```

```
}
out.println( "<br/>" );

AssetId fr_translation = mgr.getRelative( page_asset, "fr_FR" );
out.println( fr_translation );

%>
</cs:ftcs>
```

## Reading Asset and Attribute Definitions

In addition to asset data, APIs also provide access to their definitions. Information such as the attributes that make up an asset definition, type of each attribute, etc. can be obtained through a manager called `AssetTypeDefManager`. The following example attempts to read all definition information from `Document_C` and print it to the browser.

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.def.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<cs:ftcs>
<%
Session ses = SessionFactory.getSession();
        AssetTypeDefManager mgr = (AssetTypeDefManager)
              ses.getManager( AssetTypeDefManager.class.getName() );

AssetTypeDef defMgr = mgr.findByName( "Document_C", "FSII Document" );

out.println( "Asset type description: " + defMgr.getDescription() );
out.println( "<br/>" );

for( AttributeDef attrDef : defMgr.getAttributeDefs() )
{
    out.println( "Attribute name: " + attrDef.getName() );
    out.println( "Attribute description: " + attrDef.getDescription()
              );
    out.println( "is required: " + attrDef.isDataMandatory() );
    out.println( "Attribute type: " + attrDef.getType() );
    out.println( "<br/>" );
}

%>
</cs:ftcs>
```

## Reading Key-Value Mappings

The Asset API provides access to a given CSElement or Template's key-value mapping pairs. The following example reads all mapping pairs from a CSElement:

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<%@ page import="java.util.*"%>
```

```
<cs:ftcs>
<%
Session ses = SessionFactory.getSession();
AssetDataManager mgr =
              (AssetDataManager)(ses.getManager(AssetDataManager.class
              .getName()));
Condition c = ConditionFactory.createCondition("name",
              OpTypeEnum.LIKE, "FSIICommon/Nav/LocaleForm");
Query query = new SimpleQuery("CSElement", null, c, null);
query.getProperties().setReadAll(true);
for (AssetData data : mgr.read(query))
{
   List<AttributeData>  mappingArray =
              (List<AttributeData>)data.getAttributeData("Mapping").
               getData();
   for (int i=0; i<mappingArray.size(); i++)
   {
      HashMap mappingMap = (HashMap)mappingArray.get(i).getData();
      String key =
          (String)((AttributeData)mappingMap.get("key")).getData();
      String type =
          (String)((AttributeData)mappingMap.get("type")).getData();
      String value =
          (String)((AttributeData)mappingMap.get("value")).getData();
      String siteid =
          (String)((AttributeData)mappingMap.get("siteid")).getData();

      out.println("Mapping Entry #"+String.valueOf(i+1));
      out.println("<br/>");
      out.println("Key: "+key);
      out.println("Type: "+type);
      out.println("Value: "+value);
      out.println("Siteid: "+siteid);
      out.println("<br/>");
   }
}
%>
</cs:ftcs>
```

# Asset API Write

The Asset API supports write operations on basic assets and selected types of flex assets. The supported write operations are asset creation, modification, and deletion. The supported assets are basic assets, flex assets, and flex parents. All other asset types are not currently supported: Flex Parent Definition, Flex Asset Definition, Flex Filter, and Flex Attribute. An `UnsupportedOperationException` will be thrown if any write operation (insert or update) is attempted on those asset types.

Here are the topics we'll be covering in this section:

- Creating New Assets
- Updating Existing Assets
- Deleting Existing Assets
- Multilingual Assets

## Creating New Assets

Asset API uses the `AssetDataManager.insert( List<AssetData> data )` method to create a new asset in Content Server. The method takes in a list of `AssetData` and uses these `AssetData` to create new assets in Content Server. If successful, the method will populate the passed in `AssetData` with the IDs of the newly created assets.

### Creating a New Flex Asset

Asset API is able to create a new flex asset by combining the `AssetDataManager.newAssetData` method and `insert` method. The `newAssetData` method will return an empty `AssetData` with all the `AttributeData` objects populated with `null` or empty List/Collection. The `flextemplateid` (for flex assets) or `flexgroupid` (for flex parents) will be automatically populated if the correct subtype is specified.

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<%@ page import="com.openmarket.xcelerate.asset.AssetIdImpl"%>
<cs:ftcs>
<%
Session ses = SessionFactory.getSession();
AssetDataManager adm = (AssetDataManager) ses.getManager(
              AssetDataManager.class.getName() );
        MutableAssetData d = adm.newAssetData( "Content_C", "FSII
              Article" );
        d.getAttributeData( "name" ).setData( "New Content" );
        d.getAttributeData( "FSIIHeadline" ).setData( "headline" );
        d.getAttributeData( "FSIIAbstract" ).setData( "abstract" );
        d.getAttributeData( "FSIIBody" ).setData( "body" );
        d.getAttributeData( "Publist" ).setData( Arrays.asList(
              "FirstSiteII" ) );
        d.setParents( Arrays.<AssetId>asList( new AssetIdImpl(
              "Content_P", 1112192431478L)) );
        adm.insert( Arrays.<AssetData>asList( d ));
```

```
        out.println( d.getAssetId() );
%>
</cs:ftcs>
```

## Create a New Basic Asset

Asset API is also capable of creating new basic assets.

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<cs:ftcs>
<%
Session ses = SessionFactory.getSession();
AssetDataManager adm = (AssetDataManager) ses.getManager(
             AssetDataManager.class.getName() );
        MutableAssetData d = adm.newAssetData( "HelloArticle", "" );
        d.getAttributeData( "name" ).setData( "New Hello Article" );
        d.getAttributeData( "headline" ).setData( "headline" );
        d.getAttributeData( "byline" ).setData( "abstract" );
        d.getAttributeData( "category" ).setData( "g" );
        BlobObject b = new BlobObjectImpl( "filename.txt", null,
             "body".getBytes() );
        d.getAttributeData( "urlbody" ).setData( b );
        d.getAttributeData( "Publist" ).setData( Arrays.asList(
             "HelloAssetWorld" ) );
        adm.insert( Arrays.<AssetData>asList( d ));
        out.println( d.getAssetId() );
%>
</cs:ftcs>
```

# Updating Existing Assets

Another operation that Asset API supports, in addition to insert, is update. Similar to insert, update will save the data from `AssetData` into Content Server, but to an existing asset. If the asset does not exist, update will throw an exception.

```
<%@ page import="com.openmarket.xcelerate.asset.AssetIdImpl"%>
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<cs:ftcs>
<%
Session ses = SessionFactory.getSession();
AssetDataManager adm = (AssetDataManager) ses.getManager(
             AssetDataManager.class.getName() );
Iterable<AssetData> assets = adm.read( Arrays.<AssetId>asList( new
             AssetIdImpl( "HelloArticle", 1238171255471L), new
             AssetIdImpl( "Content_C", 1238171254486L)) );
        List<AssetData> sAssets = new ArrayList<AssetData>();
        for ( AssetData a : assets )
        {
```

```
            sAssets.add( a );
            a.getAttributeData( "name" ).setData( "Changed Name" );
        }
        adm.update( sAssets );
%>
</cs:ftcs>
```

## Deleting Existing Assets

AssetAPI also supports deletion of assets from Content Server. In the following example, after removing all the references to the two assets, `adm.delete` will be able to delete both assets from Content Server. An exception will be thrown if an asset is referenced by other assets, or if the asset is invalid. The delete process will stop when an exception is thrown.

```
<%@ page import="com.openmarket.xcelerate.asset.AssetIdImpl"%>
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="java.util.*"%>
<%@ page import="com.fatwire.assetapi.query.*"%>
<cs:ftcs>
<%
 Session ses = SessionFactory.getSession();
 AssetDataManager adm = (AssetDataManager) ses.getManager(
             AssetDataManager.class.getName() );
 adm.delete( Arrays.<AssetId>asList( new AssetIdImpl( "HelloArticle",
             1238171255471L), new AssetIdImpl( "Content_C",
             1238171254486L)) );
%>
</cs:ftcs>
```

## Multilingual Assets

Asset API supports the creation of multilingual assets. Creation of the assets requires a two-step process. First, the asset is created and saved. Next, locale information is added. The following example creates a new `Content_C asset` and sets the locale to `en_US`.

```
<%@ page import="com.fatwire.system.*"%>
<%@ page import="com.fatwire.assetapi.data.*"%>
<%@ page import="java.util.*"%>
<%@ page import=" com.fatwire.mda.*"%>
<%@ page
             import="com.fatwire.mda.DimensionableAssetInstance.Dimen
             sionParentRelationship"%>
<%@ page import="com.openmarket.xcelerate.asset.*"%>
<%@ page import="com.openmarket.xcelerate.common.*"%>
<cs:ftcs>

<%
Session ses = SessionFactory.getSession();
```

```
AssetDataManager adm = (AssetDataManager) ses.getManager(
             AssetDataManager.class.getName() );
MutableAssetData d = adm.newAssetData( "Content_C", "FSII Article" );
d.getAttributeData( "name" ).setData( "New Content" );
d.getAttributeData( "FSIIHeadline" ).setData( "headline" );
d.getAttributeData( "FSIIAbstract" ).setData( "abstract" );
d.getAttributeData( "FSIIBody" ).setData( "body" );
d.getAttributeData( "Publist" ).setData( Arrays.asList( "FirstSiteII"
             ) );
d.setParents( Arrays.<AssetId>asList( new AssetIdImpl( "Content_P",
             1112192431478L)) );
adm.insert( Arrays.<AssetData>asList( d ));
DimensionManager dam =
             (DimensionManager)ses.getManager(DimensionManager.class.
             getName());
Dimension dim = dam.loadDimension("en_US");
d.getAttributeData("Dimension") .setData(Arrays.asList(new
             Dimension[]{dim}));
DimensionParentRelationship dpr = new
             DimParentRelationshipImpl("Locale", d.getAssetId());
d.getAttributeData("Dimension-parent").setData(Arrays.asList(new
             DimensionParentRelationship[]{dpr}));
adm.update( Arrays.<AssetData>asList( d ));
%>
</cs:ftcs>
```

# Development Strategies

Topics in this section are the following:

- Data Types and Attribute Data (maps CS data types to Java types)
- Query Types (compares types of queries, their usage, and supported operations)

## Data Types and Attribute Data

`AttributeData` contains information about the data type (Content Server specific type) and the actual data. The types are defined by `AttributeTypeEnum`. `AttributeData.getData()` and `AttributeData.getDataAsList()` return data objects of a specific Java type. Here is a map of Content Server types and their corresponding Java types.

**Table 17:** CS Data Types and Java Types

| CS Data Type | Java Type |
|---|---|
| INT | Integer |
| FLOAT | Double |
| STRING | String |
| DATE | Date |
| MONEY | Double |
| LONG | Long |
| LARGE_TEXT | String |
| ASSET | AssetId |
| BLOB | BlobObject |

## Query Types

Using the Asset API, you can perform two kinds of queries: generic/basic and flex.

There are two different algorithms, one using the generic asset infrastructure (generic/basic query), and the other using AssetSets and Search States (flex query). Note that it is possible to use the generic/basic query for flex assets as well as basic assets; flex query, however, works only for flex assets.

Each of these algorithms has its advantages and disadvantages. The Asset API seeks to unify the querying mechanism and eventually let the API user not be concerned about the choice of algorithm. However, at the present time as there is no equivalence between these algorithms, the user needs to specify if she wants to use a specific feature, offered by one of the two.

`QueryProperties.setIsBasicSearch( true )` sets the query algorithm to generic/basic search for this query. It is set to `false` by default. For basic assets, the setting does not matter.

## Which Type of Query to Choose?

Very simply put, if you want to look for basic attributes of a flex asset, use the basic/

. Otherwise use the default. This will work for most queries one generally encounters.

Things are a bit more subtle than that. Given below are other considerations against each type of query.

### Basic/Generic Query

- Cannot search on flex attributes if you do not specify subtype.
- Cannot search on a flex attribute that is not in the flex definition.
- Cannot sort on a flex attribute.
- Case sensitivity is not guaranteed (depends on the database).
- Only the `AND` operation is allowed between different fields (`name=name1 AND description=descr1` is allowed, but `name=name1 AND name=name2` is not).
- Only `OR` is allowed for two conditions involving the same field name. The `OR` condition does not work on flex attributes.

### Flex Query

- Cannot have basic attributes in the condition (`id`, name, description, etc.).
- Cannot sort by basic attributes.
- Flex query works without a subtype being specified. The search applies to data of all subtypes.
- Can use only the following operands in the condition; `LIKE`, `EQUALS`, `BETWEEN`, and `RICHTEXT`.

## Data Types and Valid Query Operations

Depending on the type of query being performed, there are further restrictions on what type of operation is allowed for a given data type.

In general, a flex type query (which is the default for flex assets) allows only the following OpTypeEnums; `LIKE`, `EQUALS`, `BETWEEN`, and `RICHTEXT`. Note that these are the same operations available from `AssetSet/SearchState` tags.

If you want to use other OpTypeEnums, you have to use basic/generic query (by setting `QueryProperties.setIsBasicSearch( true )`). Such a query, of course, has to adhere to the basic query rules above.

Here is the allowed set of operations per data type (single-valued or multi-valued) for a basic/generic query.

**Table 18:** Operations Permitted in Basic/Generic Query

| Data Type | EQUALS | NOT_EQUALS | LIKE | GREATER | LESSTHAN | BETWEEN | RICHTEXT |
|---|---|---|---|---|---|---|---|
| INT | Y | Y | N | Y | Y | – | N |
| FLOAT | Y | Y | N | Y | Y | – | N |
| STRING | Y | Y | Y | Y | Y | – | N |
| DATE | Y | Y | N | Y | Y | – | N |

**Table 18:** Operations Permitted in Basic/Generic Query (continued)

| Data Type | EQUALS | NOT_EQUALS | LIKE | GREATER | LESSTHAN | BETWEEN | RICHTEXT |
|---|---|---|---|---|---|---|---|
| MONEY | Y | Y | N | Y | Y | – | N |
| LONG | Y | Y | N | Y | Y | – | N |
| LARGE_TEXT | N | N | Y | N | N | – | N |
| ASSET | Y | Y | N | N | N | – | N |
| BLOB | N | N | N | N | N | – | N |

And here is the allowed set of operations per data type (single-valued or multi-valued) for the flex type query.

**Table 19:** Operations Permitted in Flex Query

| Data Type | EQUALS | NOT_EQUALS | LIKE | GREATER | LESS THAN | BETWEEN | RICHTEXT |
|---|---|---|---|---|---|---|---|
| INT | Y | – | N | – | – | Y | N |
| FLOAT | Y | – | N | – | – | Y | N |
| STRING | Y | – | Y | – | – | Y | N |
| DATE | Y | – | N | – | – | Y | N |
| MONEY | Y | – | N | – | – | Y | N |
| LONG | Y | – | N | – | – | Y | N |
| LARGE_TEXT | N | – | Y | – | – | N | Y |
| ASSET | Y | – | N | – | – | Y | N |
| BLOB | N | – | N | – | – | N | Y |

# Optional: Setting Up to Use the Asset API from Standalone Java Programs

Although this tutorial shows usage of the Asset API from JSP templates, it is possible to use the API from a standalone Java program, as well. In order to do this, you can set up a single database connection or a connection pool (outside Content Server).

**Note**

In the steps below, we assume that all components are local to your Content Server installation.

### Before setting up a single database connection or pool

1. Make sure the following files are in the classpath of your Java program:
   - `javaee.jar` and `tools.jar` (both are available in JDK 1.5 and higher versions).
   - `ServletRequest.properties`. This file can be copied from the `WEB-INF/classes` folder.
   - `wem_sso.xml`, if your Content Server installation has WEM installed and single sign-on (in `futuretense.ini`) is set to `true`. The `wem_sso.xml` file can be copied from the `WEB-INF/classes` folder.
   - All of Content Server's binary files (`jar` files in the `WEB-INF/lib` folder).
2. Continue with the steps in one of the following sections:
   - To set up a single database connection
   - To set up a database connection pool

### To set up a single database connection

1. Set the following system properties for your Java program:

```
cs.dburl=<JDBC_URL_to_connect_to_DB>
cs.dbdriver=<driverClass>
cs.dbuid=<dbUserName>
cs.dbpwd=<dbPassword>
```

2. Locate the Content Server installation folder and pass its name as a JVM argument:

```
-Dcs.installDir=<install_dir>
```

### To set up a database connection pool

1. Add the following `jar` files to the classpath: `commons-dbcp.jar` (which ships with Content Server) and `commons-pool.jar` (available from the Apache website).
2. Create the property file with the same name as your data source:

   **Note the following:**

   - The name of the data source is the value of the `cs.dsn` property (in `futuretense.ini`).

- The value of `cs.dbconnpicture` (also in `futuretense.ini`) must refer to `cs.dsn` (the combination of `cs.dbconnpicture` and `cs.dsn` must yield a valid resource).

  For example, the combination of

  `cs.dsn=` `csDataSource` *and*

  `cs.dbconnpicture=` `java\:comp/env/$dsn`

  yields a valid resource:

  `java:/csDataSource`

  Therefore, you would name the property file `csDataSource.properties` (the value of `cs.dsn`).
- When creating the property file, make sure to place it in the classpath of your Java program.

3. When the property file is created, add the following keys to the file:

```
driver=<driverClass>
url=<JDBC_URL_to_connect_to_DB>
maxconnections=<number_of_connections_to_pool>
user=<dbUserName>
password=<dbPassword>
```

4. Locate the Content Server installation folder and pass its name as a JVM argument:

```
-Dcs.installDir=<install_dir>
```

Chapter 44

# Public Site Search

Content Server 7.5 introduces a new framework for managing search indices. This framework forms the basis for searches in both the editorial interface (Dash) and on the live site – that is, the visitors' side. Hence the name "Public Site Search."

This chapter introduces the search framework and discusses the usage of the API in building public site searches.

This chapter contains the following sections:


- Overview of the Search Framework
- Index Types
- Search API Overview
- Advanced Configuration

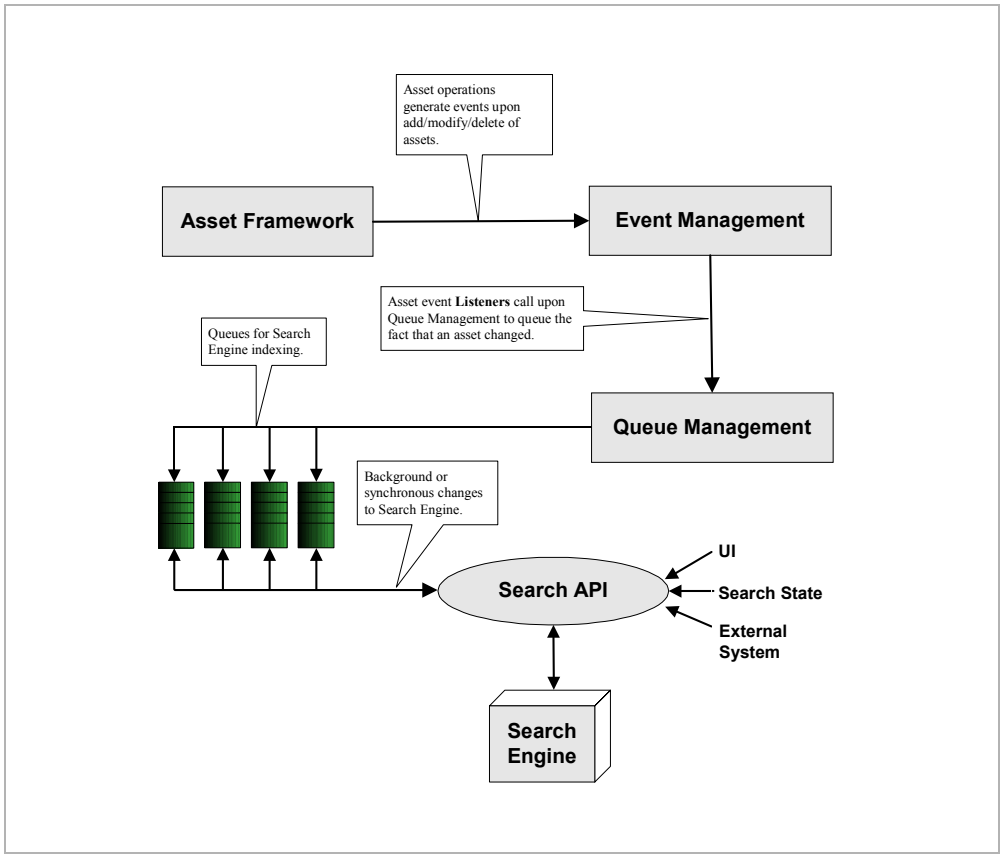
# Overview of the Search Framework

**Note**

You can skip this section if you are primarily interested in the usage of the search API.

Figure 15 shows how the search engine integration framework works with the rest of Content Server.

**Figure 15:** Search Engine Integration




The search framework consists of the Search API, special asset event listeners, and a polling system for queues. This framework is used in coordination with the Event Management and Queue Management frameworks. This document focuses primarily on the search framework.

1. Asset framework detects changes/additions to assets and fires off events.
2. Registered listeners queue the changes, using a persistent queue implementation. A given event can be queued into one or many persistent queues. Each queue can be thought of as the source of data for a search index.
3. Once asset events are queued, a background process empties the queue contents and routes them to the Search API.

4. The Search API chooses the appropriate (configurable) search engine vendor implementation to start the indexing process.

# Index Types

Two types of indices are created in Content Server: `Global` index and `AssetType` index. `Global` index is the index of all data (all asset types enabled for `Global` index). If you want to search for a phrase or expression in multiple asset types (such as attempting to build a Google-like search interface), `Global` index is more appropriate.

While `Global` index contains data for all fields of the index, it does not store the data in a form that is suitable for parametric searches. An `AssetType` index contains indexed information for a given asset type in a manner that can be searched parametrically. (The Content Server Advanced interface supports the configuration of Asset Type searches, which includes attribute-based searches for the indexing-enabled asset types. More information about the search configuration options is available in the *Content Server Administrator's Guide*.)

## Global Index

`Global` index is used by the Dash interface to build a global search UI. An instance of the index also exists on the delivery server. The index on the delivery server can be used to build public site searches.

Note that only those assets that are published to the live site *after search is configured* are available for searches. It is during publishing that the data gets indexed. All assets that may exist on the live site before search is configured will not be reflected in the `Global` index (until the assets are reindexed on the live site).

A search index functions roughly similar to a database table. `Global` index consists of the following fields:

**Note**

The field names are case sensitive.

**Table 20:** Fields in the Global Index

| Field name | Description |
|---|---|
| `defaultSearchField` | This field contains all the data of the indexed asset. This is the field you would search for in full-text searching.<br>The `defaultSearchField` contains index data for all attributes of the asset, including any binary field data. Data for all the attributes is merged into this single field and indexed. The index itself does not 'store' data for this field, but does allow full-text searching. |
| `id` | This field contains the asset id. |
| `AssetType` | This field contains the asset type (Content_C/Product_P). |
| `locale` | This field contains the locale string (for example, `en_US`). |

**Table 20:** Fields in the Global Index (continued)

| Field name | Description |
|---|---|
| `name` | This field contains the name of the asset. |
| `description` | Description associated with the asset. |
| `subtype` | Name of the subtype (flex definition name). |
| `subtypeid` | ID of the subtype (flex def id). |
| `updateddate` | Last updated date as found at the time of indexing. |
| `siteid` | IDs of all sites in which this asset is available. |
| `startdate` | Start date field in the asset table. |
| `enddate` | End date field in the asset table. |

## Asset Type Index

An asset type index is created when it is enabled from the Advanced interface (**Admin tab > Search > Configure Asset Type search**). Once an asset type is enabled, an index will be created under `/shared/lucene/<Asset type name>`. This index contains all attributes of the given type as fields in the index. For example, the index for `Content_C` contains the following fields:

**Note**

The field names are case sensitive.

**Table 21:** Fields in the Asset Type Index

| Field Name | Description |
|---|---|
| `DefaultSearchField` | This field contains all the data of the indexed asset. This is the field you would search for in full text searching.<br>The `DefaultSearchField` contains index data for all attributes of the asset, including any binary field data. Data for all the attributes is merged into this single field and indexed.<br>The index itself does not 'store' data for this field, but does allow full text searching. |
| `id` | This contains the asset id |
| `AssetType` | This contains the asset type (Content_C/Product_P) |
| `locale` | Contains the locale string (example en_US) |
| `name` | Name of the asset |
| `description` | Description associated with the asset |
| `subtype` | Name of the subtype (flex definition name) |
| `subtypeid` | Id of the subtype (flex def id) |
| `updateddate` | Last updated date as found at the time of indexing. |
| `siteid` | All site ids this asset is available in |

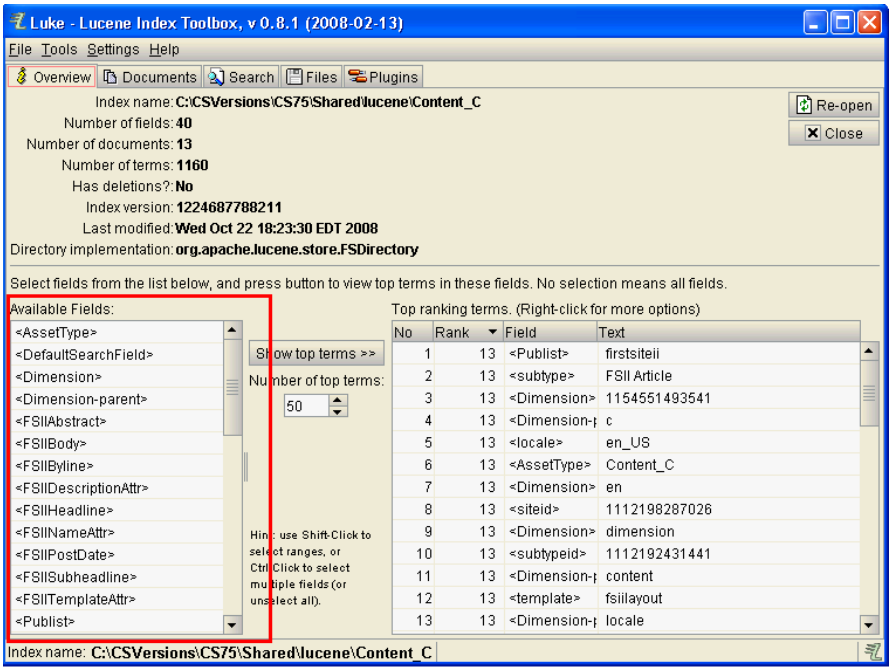
**Table 21:** Fields in the Asset Type Index (continued)

| Field Name | Description |
|---|---|
| startdate | Startdate field in the asset table |
| enddate | Enddate field in the asset table |
| Dimension | ID of the dimension |
| Dimension-parent | ID of the Dimension parent |
| createdby | User name that created this asset |
| createddate | Date the asset was created |
| Publist | List of site names this asset belongs to |
| Relationships | Asset IDs of related items (flex only) |
| externaldoctype | Not used |
| filename | File name used for static publishing |
| flextemplateid | ID of the flex definition (flex only) |
| fw_uid | Globally unique id of this asset |
| path | Path used for static publishing |
| renderid | |
| ruleset | XML document of the ruleset |
| status | Status associated with the asset |
| subtype | Subtype name |
| subtypeid | Subtype id (flex only) |
| template | Template name |
| updatedby | User name that last updated this asset |
| updateddate | Date of last update |
| urlexternaldoc | Not used |
| urlexternaldocxml | Not used |
| FSIIAbstract | Flex attribute |
| FSIIBody | Flex attribute |
| FSIIByline | Flex attribute |
| FSIIDescriptionAttr | Flex attribute |
| FSIIHeadline | Flex attribute |
| FSIINameAttr | Flex attribute |
| FSIIPostDate | Flex attribute |
| FSIISubheadline | Flex attribute |
| FSIITemplateAttr | Flex attribute |

To visualize which fields are available in a given index, use the tool Luke, available at:

`http://www.getopt.org/luke/`

Once you launch the tool, use the tool's browse function to load the index by simply locating the folder that contains the index (for example: `../shared/lucene/Content_C`). The figure below shows the list of fields in Luke.



# Search API Overview


- SearchEngine
- QueryExpression
- Basics of the Search API
- Configuring Lucene Parameters


## SearchEngine

The `SearchEngine` interface defines the key functions of a search engine implementation; indexing and searching. (More information about SearchEngine is available in *Javadoc*.)

- The source of index data is given to `SearchEngine`. `SearchEngine`, in response to the indexing request, creates the search index, if it is not already created and updates the contents based on `IndexSource's` accessors.
- `index()` works off of a given `IndexSource` instance. Depending on the search engine's implementation details, it operates on new, modified and deleted data coming from the `IndexSource` (in most search engines, all that is modified must be deleted first and then re-indexed).

  `index()` also invokes index lifecycle methods (`startIndexing()` and `endIndexing()`) on the given instance of `IndexSource`.
- `search()` operates on a `QueryExpression` against one or many indexes (`IndexSources`), resulting in a single set of results, sorted by their relevance (or `SortOrder`, if specified and usable across indices.
- A configuration lookup interface (`IndexSourceConfig`) is supplied to the `SearchEngine` instance which it can look up `IndexSource` properties, if needed.
- A `QueryConverter` interface is supplied to `SearchEngine`. This interface converts a given `QueryExpression` to its native form (recognizable by the specific search engine). This makes it possible to control the query language that the search engine uses externally.
- `SearchResult` is an abstraction over what is returned from the search engine. `SearchResult` is an iterator over `ResultRow`, a sub class of `IndexRow`, that contains relevance information. The `getRelavence()` method returns a double; the higher the value, the higher is the relevance of this `ResultRow` for the given query.

## QueryExpression

The `QueryExpression` interface is a generic interface for defining search criteria. All search engines support native formats for building queries. The native form contains definitions of wildcards, relevance hints, etc. These tend to be very specific to each search engine.

Search engines also provide a basic query construct, which can be programmatically built (`AND & OR` over field matches). These can be thought of in terms of generalized programmable interfaces, although limited in power.

`QueryExpression` encapsulates four distinct characteristics of search engine queries:

- Native text search format: Most search engines support a very sophisticated native format for search, including wild cards, special hints etc. This is available through `getStringFormat()`.

- Conjunction and disjunction: ANDs and ORs of conditions using `and()` and `or()` methods.
- Pagination: Using `getStartIndex()` and `getMaxResults()` methods
- Sorting: using `getSortOrder()`

## Configuration

1. Make sure that search indexing is enabled:

   In the `SystemEvents` table, verify that the `SearchIndexEvent` is enabled (enabled field =1). This is configured to run in the background constantly (`*:*:* */*/*`); in practice it runs about every 30 seconds.

2. Make sure Asset listener is registered:

   Assets are queued for processing by the search framework, using asset events. The asset events are registered in the `AssetListener_reg` table. Make sure the following entry is found in this table (if it does not exist, add it).

| ID | Listener | Blocking |
|---|---|---|
| 1153937286234 | com.openmarket.basic.event.SearchAssetIdEventListener | Y |

`IndexSourceMetaDataConfig` – table that stores configuration information for `IndexSource`. This describes the structure and nature of the index itself. This should have a row for '`Global`' by default. Any asset type enabled for Asset type index will have an additional row in this table.

`SearchEngineMetaDataConfig` – stores the search engine configuration. This table should have a row for Lucene, by default.

These are configured correctly by the installer and managed by the Admin interface.

Defaults here should suffice; the precise definition and meaning of "configuration" will be covered in some depth at a later point in this tutorial (see "Advanced Configuration," on page 848).

## Basics of the Search API

Suppose you want to search the `Global` index for all assets that contain the word 'Audio' and they are of type `Content_C`. Here are the steps you would follow:

1. Get a handle to the `SearchEngine` interface and metadata.
2. Build a `QueryExpression`.
3. Perform the search and read the results.

### Step 1:

```
IndexSourceConfig con = SearchUtils.getIndexSourceConfig();
IndexSourceMetadata metadata = con.getConfiguration( "Global" );
String engineName = metadata.getSearchEngineName();
SearchEngineConfig searchConfig =
SearchUtils.getSearchEngineConfig();
SearchEngine eng = searchConfig.getEngine( engineName );
```

We will need to get a handle to `IndexSourceConfig`. This interface provides lookup functionality for configuration information for `IndexSources` (an `IndexSource` roughly represents an index). The helper class `SearchUtils` provides convenience methods for getting `IndexSourceConfig`.

Once we have `IndexSource`, we will look up the metadata for `Global` index (line 2) and the name of the engine configured for it (line 3). To get an instance of `SearchEngine` using the engine name, we need a `SearchEngineConfig` handle (line 4 and 5).

### Step 2:

```
QueryExpression q = SearchUtils.newQuery(
metadata.getDefaultSearchField(), Operation.CONTAINS, "Audio" );
q = q.and("AssetType", Operation.EQUALS, "Content_C");
q.setMaxResults(1000);
```

`QueryExpression` is a construct that can be nested with `AND`s and `OR`s. In this example, we are creating the first condition and performing an `and()` operation on it with the second condition. **Note that the result of *and()* operation is assigned back to *q*.** The `and()` or `or()` operation does not change the `QueryExpression` it is invoked on; it returns a new instance, which is the result of the operation. We also `setMaxResults()` to limit the number of results to 1000.

Alternatively you could build a generic search expression string (with valid wild cards, etc., as specified in: `http://lucene.apache.org/java/docs/queryparsersyntax.html`). You would use:

```
QueryExpression q = SearchUtils.newQuery("your expression");
```

### Step 3:

Now, we want to query the `SearchEngine` and read the results.

```
SearchResult<ResultRow> res = eng.search( Collections.singletonList(
"Global" ), q );for(;res.hasNext();)
{
       ResultRow row = res.next();
       // In each row, let's look for the AssetId and type
       IndexData idData = row.getIndexData("id");
       IndexData typeData = row.getIndexData("AssetType");
       Long id = Long.parseLong(idData.getData());
       String type = typeData.getData();
       Double rel = row.getRelevance();
       ….
}
```

The result from `SearchEngine.search()` is a special `Iterator` called `SearchResult`. In addition to Iterator methods, this interface supports `getTotal()`, whose result indicates the total number of results obtained.

Each element of `SearchResult` is a `ResultRow`. A `ResultRow` contains data from the index as well as the relevance (`getRelevance()`) or score indicating how well the contents of this row match the search criteria. `ResultRow` also contains all data in the index; you can look them up by getting an `IndexData` for each field, as shown above.

Once you have the asset ID and asset type, you can build a URL for the template and show these as search results. You can an also use the relevance to indicate the match level, as done in the Dash interface.

# Advanced Configuration

The following sections explain aspects of advanced configuration that may be necessary in certain circumstances. However, in most cases, the defaults should be sufficient.

## Configuring Lucene Parameters

**Note**

Some of these parameters can cause significant changes in the way the index performs at run time. Check Lucene's documentation and rely on experimentation to determine the best settings for your site. If you do not have a compelling reason to change the defaults, it is highly advised that you keep the defaults.

In the Lucene search engine, an index can be created with a certain set of parameters that determine how the index is created and how it performs. While Lucene's default parameters are reasonable, Content Server provides administrators with a way to change them.

The `IndexSourceMetaDataConfig` table contains one row per index you would find in the system. Each row has a field named "`properties`" whose contents are used to configure Lucene parameters. Parameter-value pairs are separated by a semicolon ( ';' ) as shown below:

*`param1=value1;param2=value2`*

Parameters supported by Content Server are listed in Table 22, on page 849.

**Table 22:** Parameters supported by Content Server

| Parameter | Type | Description |
|---|---|---|
| `mergeFactor` | Integer | Determines how often segment indices are merged.<br>With smaller values, less RAM is used while indexing, and searches on unoptimized indices are faster, but indexing speed is slower.<br>With larger values, more RAM is used during indexing, and while searches on unoptimized indices are slower, indexing is faster. Thus larger values (> 10) are best for batch index creation, and smaller values (< 10) for indices that are interactively maintained.<br>This must never be less than `2`. The default value is `10`. |
| `maxMergeDocs` | Integer | Determines the largest number of documents ever merged. Small values (e.g., less than 10,000) are best for interactive indexing, as this limits the length of pauses while indexing to a few seconds. Larger values are best for batched indexing and speedier searches.<br>Defaults to max integer value ($2^{31}$-1). |
| `maxBufferedDocs` | Integer | Determines the minimal number of documents required before the buffered in-memory documents are merging and a new Segment is created.<br>Defaults to `10`. |
| `optimizeInterval` | Integer | Determines the time interval (in seconds) between `optimize()` calls. The default value is `30` seconds, which is the recommended value for most systems. If you have a large amount of data changes, set this parameter to any value within the range of 300–600 seconds. |
| `commitLockTimeout` | Long | Sets the maximum time to wait for a commit lock (in milliseconds).<br>Defaults to `10000`. |
| `maxFieldLength` | Integer | Maximum number of terms that will be indexed for a single field in a document. This limits the amount of memory required for indexing, so that collections with very large files will not crash the indexing process by running out of memory. |

**Table 22:** Parameters supported by Content Server (continued)

| Parameter | Type | Description |
|---|---|---|
| `maxFieldLength` *(continued)* | Integer | Note that this effectively truncates large documents, excluding from the index terms that occur further in the document. If you know your source documents are large, be sure to set this value high enough to accommodate the expected size. If you set it to max value of Integer ($2^{31}$-1), then the only limit is memory, but you should anticipate an OutOfMemoryError.<br>By default, no more than 10,000 terms will be indexed for a field. |
| `termIndexInterval` | Integer | Sets the interval between indexed terms. Large values cause less memory to be used by IndexReader, but slow random-access to terms. Small values cause more memory to be used by an IndexReader, and speed random-access to terms.<br>This parameter determines the amount of computation required per query term, regardless of the number of documents that contain that term. In particular, it is the maximum number of other terms that must be scanned before a term is located and its frequency and position information may be processed. In a large index with user-entered query terms, query processing time is likely to be dominated not by term lookup but rather by the processing of frequency and positional data. In a small index or when many uncommon query terms are generated (e.g., by wildcard queries) term lookup may become a dominant cost. In particular, `numUniqueTerms/interval` terms are read into memory by an IndexReader, and, on average, `interval/2`terms must be scanned for each random term access.<br>Default value is `128`. |
| `useCompoundFile` | String (must be `yes` or `no`) | Setting to turn on usage of a compound file. When on, multiple files for each segment are merged into a single file once the segment creation is finished. |
| `writeLockTimeout` | Long | Sets the maximum time to wait for a write lock.<br>Default value is `1000`. |

## Configuring the Custom AnalyzerFactory

In Lucene, an Analyzer represents a policy for extracting index terms from text. Analyzers are used at the time of indexing as well as searching for various tasks such as removing stop words, removing white spaces, etc.

Different Analyzers exist in the Lucene repository for handling various locales. Often Analyzers are used for injecting synonyms or addressing accented characters gracefully. You can also build your own Analyzer by using any of the Lucene's standard analyzers as a basis.

Content Server's Lucene implementation uses `StandadAnalyzer`, a general purpose analyzer for the English language. However, Content Server supports custom Analyzers via a plugin interface, `AnalyzerFactory`. The configured `AnalyzerFactory` is used to look up the analyzer, when required, in the process of indexing or searching. The `AnalyzerFactory` looks up the analyzer in the following instances:

- When building the index as a whole
- When parsing a query
- When indexing an individual row

To plug in a custom `AnalyzerFactory`, you need to implement and register the `AnalyzerFactory` interface. Registration is done by modifying a row in `SearchEngineMetaDataConfig` table. Add the following to the `properties` field of the row whose "Name" field is set to `Lucene`.

```
AnalyzerFactory=<fully qualified class name of your custom
   AnalyzerFactory>
```

# Appendices

This part contains the following appendices:


- Appendix A, “Creating a Hierarchical Flex Family”
- Appendix B, “Content Server URL Assemblers”
- Appendix C, “White Space and Compression”
- Appendix D, “Asset and Publish Events in Content Server”

Appendix A

# Creating a Hierarchical Flex Family

This appendix provides a tutorial to help you create a flex family, a simple three-level hierarchy that is based on single-valued definitions (and for the time being, ignores attributes and their inheritance). When you complete the tutorial, you will have a basic understanding of the flex asset model and how it is used to create hierarchical content in Content Server. You will also have a model from which to create similar hierarchies.

This appendix contains the following sections:


- Overview
- Procedures
- Next Steps

# Overview

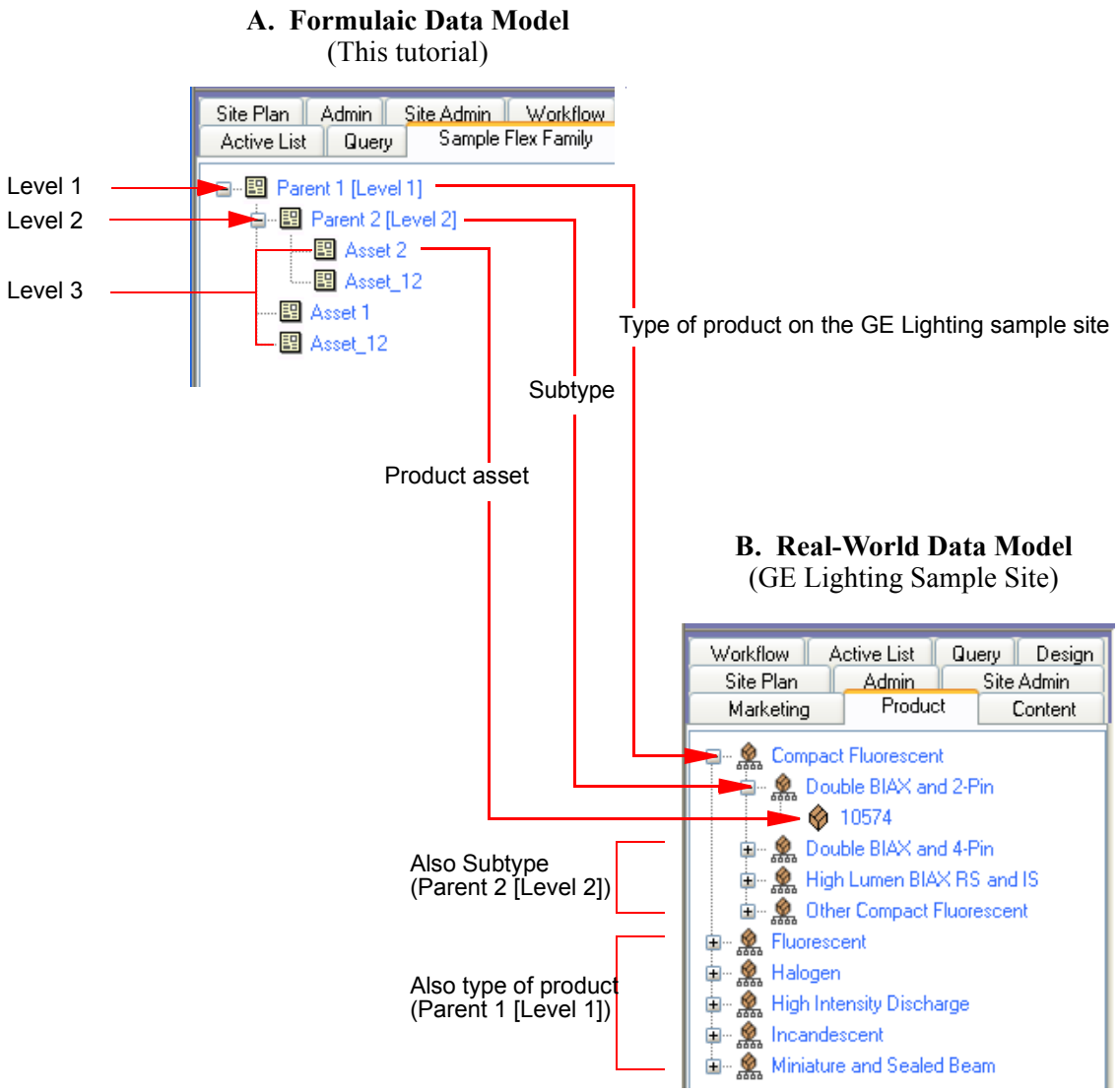
This section describes the flex family that you will be creating.

## Hierarchical Organization

The flex family that you will be creating in this tutorial consists of three levels:

- A top-level parent (named Parent 1 [Level 1] in our example).
- A second-level parent (named Parent 2 [Level 2] in our example).
- Assets, at the third level. One asset is placed directly under its level 1 parent; another asset is placed directly under its level 2 parent; the third asset is placed under both the level 1 parent and the level 2 parent.

In the Content Server interface, the hierarchy looks exactly as shown in Figure A. The representation is formulaic; Figure B shows how it translates to a real-world model, represented by the GE Lighting sample site.



On the sample site (Figure B),

- "Parent 1 [Level 1]" is named "Compact Fluorescent," a type of lighting product.
- "Parent 2 [Level 2]" is named "Double BIAX and 2-Pin," a subtype of lighting product (under "Compact Fluorescent").
- "Asset 2" is named "10574", a product asset of the subtype "Double BIAX and 2-Pin."
- Asset 1 does not exist on the GE Lighting site (no asset was placed under level 1, "Compact Fluorescent"). The same holds for Asset_12 (no asset was placed under both levels 1 and 2).

## Flex Family Specifications

Table A-1 lists the flex family members that you will be creating in this tutorial. Note that flex filters are optional components; they are not included in this tutorial

**Table A-1:** Flex Family Members

| Flex Family Member | Name | Instances | Based on Parent Definition | Based on Flex Definition |
|---|---|---|---|---|
| Flex Attribute Type | My Attribute | Attribute_1[a]<br>Attribute_2 | | |
| Flex Parent Definition Type | My Parent Definition | Level 1 Def<br>Level 2 Def | —<br>Level 1 Def | |
| Flex Definition Type | My Flex Definition | Flex Def 1<br>Flex Def 2<br>Flex Def_12 | Level 1 Def<br>Level 2 Def<br>Level 1 Def *and* Level 2 Def | |
| Flex Parent Type | My Parent | Parent 1 [Level 1]<br>Parent 2 [Level 2] | Level 1 Def<br>Level 2 Def | |
| Flex Asset Type | My Asset | Asset 1<br>Asset 2<br>Asset_12 | | Flex Def 1<br>Flex Def 1<br>Flex Def_12 |
| Flex Filter Type | — | — | — | — |

a. Suffixes 1, 2 and _12 refer to levels of the hierarchy ( "_12" denotes both levels 1 and 2). For example, "Asset 1" denotes an asset that is placed under level 1. "Asset_12" denotes an asset that is placed under both levels 1 and 2.

# Procedures

In this tutorial, you will be creating a small flex family with generic names for its family members. This approach will help you visualize the "formula" for building hierarchies.

At the end of this tutorial, you will change the names of selected family members to real-world names to understand how a formulaic data model translates to a business-related data model. You will also add more parents and assets to the hierarchy, giving them real-world names as you create them.

## Step 1: Create a Flex Family

In this step, you will create a flex family by naming its required members.

### To create the flex family

1. Open your browser and enter the address:

   `http://your_server/Xcelerate/LoginPage.html`

2. Enter your login name and password and click **Login**.
3. Select the site for the flex family (**HelloAssetWorld** in our example).
4. On the **Admin** tab, expand **Flex Family Maker** and double-click **Add New Family**.
5. In the "Flex Family Maker" form, fill in the fields exactly as shown below:

| Field Name | Enter | Comments |
|---|---|---|
| Flex Attribute | MyAttribute | In this step, you are naming the database tables that CS will create for the flex family. The names must not contain spaces. |
| Flex Parent Definition | MyParentDefinition | |
| Flex Definition | MyFlexDefinition | |
| Flex Parent | MyParent | |
| Flex Asset | MyAsset | |
| Flex Filter | | |

6. Click **Continue**.
7. In the next form, fill in the fields for each new member of the family as follows:
   a. Click in the **Description** field and enter the same name as in step 2 above, but separate the words in the name with spaces.

      The name that you enter will be used throughout the CS interface to identify the asset type.

   b. Click in the **Plural** field and enter the plural form of the name used in the preceding step.
   c. Click **Add New Flex Family**.
8. Wait for Content Server to create the flex family and return the message indicating that the flex family members (asset types) were successfully installed.
9. Go to the next step.

## Step 2: Enable the New Flex Asset Types

In this step, you will enable the flex family members for the HelloAssetWorld sample site. You will also create start menu items for the members, so that you can create and search for their instances in subsequent steps.

**Note**

If the HelloAssetWorld sample site is not installed on your system, you can enable the flex family for a site of your choice.

**To enable the new flex asset types**

1. On the **Admin** tab, expand the **Sites** node and complete the following steps:
   a. Expand **HelloAssetWorld** (the sample site where the flex family will be enabled), or a site of your choice.
   b. Under that site, expand **Asset Types** and double-click **Enable**.
      1) From the list, select the asset types that you just created (MyAsset, MyAttribute, MyFlexDefinition, MyParent, MyParentDefinition).
      2) Click **Enable Asset Types**.
   c. In the “Enable Asset Types” form:
      1) Make sure that all Start Menu options are selected (so that, later, you can create and search for instances of the family members.)
      2) Click **Enable Asset Types**.
2. Wait for Content Server to display the message indicating that the asset types have been enabled for the site.
3. Go to the next step.

## Step 3: Add a "Flex Family" Tab to Content Server's Tree

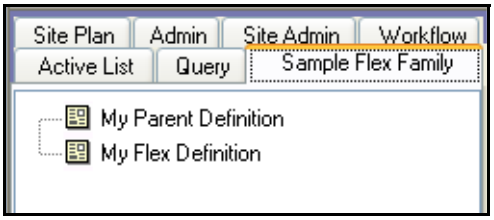
In this step, you will add a tab that tracks the creation of your flex family. You will set up this tab to display selected members of the flex family as you finish creating them.

### To add the tree tab

1. On the **Admin** tab, double-click the **Tree** node.
2. In the "Tree Tabs" form, scroll to the bottom and click **Add New Tree Tab**.
3. In the "Add New Tree Tab" form, fill in the fields as follows:

| Field Name | Enter or Select | Comments |
|---|---|---|
| Title | Sample Flex Family | Name of the tab. |
| Tooltip | Sample Flex Family | Description of the tab. |
| Sites | HelloAssetWorld<br>(or the site you chose in Step 2: Enable the New Flex Asset Types) | Site on which to enable the flex family. |
| Required Roles | Any | |
| Tab Contents | My Parent Definition<br>My Parent<br>My Flex Definition<br>My Asset<br><br>**Note:** Click **Add Selected Items** and use the **Display Order** arrow to arrange the members in the order shown above. | |

4. Click **Save**.
5. Refresh the screen.
6. Click the "Sample Flex Family" tab and make sure its contents are identical to the display below:




7. Go to the next step.

## Step 5: Create Parent Definition Assets

In this step, you will create two parent definitions. The first parent definition establishes the top level of the hierarchy; the second parent definition establishes the second level.

**To create the first parent definition asset**

1. From the button bar, click **New**.
2. From the list of options that appears, select **New My Parent Definition**.
3. In the next form:
   a. Fill in the fields as follows:

| Field Name | Enter or Select | Comments |
|---|---|---|
| Name | Level 1 Def | This is our name for the definition of the first level of the hierarchy. |
| Description | Level 1 Def | |
| Parent Select Style | Select Boxes | |
| My Parent Definitions | | No parent definitions are selected (or available) in this field. Therefore, this parent definition establishes the first level of the hierarchy. |

   b. Click **Save**.

**To create the second parent definition asset**

1. From the button bar, click **New**.
2. From the list of options that appears, select **New My Parent Definition**.
3. In the next form:
   a. Fill in the fields as follows:

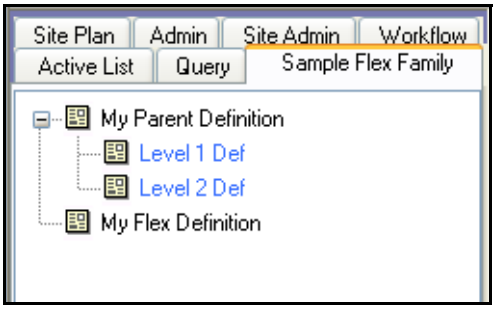
| Field Name | Enter or Select | Comments |
|---|---|---|
| Name | Level 2 Def | This is our name for the definition of the second level of the hierarchy. |
| Description | Level 2 Def | |
| Parent Select Style | Select Boxes | |

| Field Name | Enter or Select | Comments |
|---|---|---|
| My Parent Definitions | Level 1 Def<br>**Note:** Under "Single Value," click the **Required** arrow to move your selection to the "Selected" list box. | Choosing **Level 1 Def** subordinates the current parent definition to **Level 1 Def**. Chaining definitions in this manner establishes Level 2 Def as the second level.<br>When parents are created and based on the current parent definition (Level 2 Def), they are subordinated to parents that are based on Level 1 Def. |

**b.** Click **Save**.

**4.** Refresh the screen.

**5.** Click the "Sample Flex Family" tab and expand its contents to make sure they are identical to the display below:




**6.** Go to the next step.

## Step 6: Create Flex Parent Assets

In this step, you will create two flex parent assets, and base them on the flex parent definitions that you created in the previous step. The first parent asset will occupy the top level of the hierarchy. The second parent asset will occupy the second level of the hierarchy.

### To create the top-level parent of the hierarchy

1. From the button bar, click **New**.
2. From the list of options that appears, select **New My Parent**.
3. In the form that appears, fill in the fields as follows:

| Field Name | Enter or Select | Comments |
|---|---|---|
| Name | Parent 1 [Level 1] | This is our name for a level 1 parent in the hierarchy (in our example, the name is genericized simply to help you identify the level).<br>**Note:** At the end of this tutorial, you will change the name to a business-specific name (“Outdoor Sports Equipment,” in our example, which describes the inventory of a company dealing with sports gear.) |
| My Parent Definition | Level 1 Def | Choosing **Level 1 Def** places the parent you are creating at the top level of the hierarchy. |

4. In the next form, click **Save**.

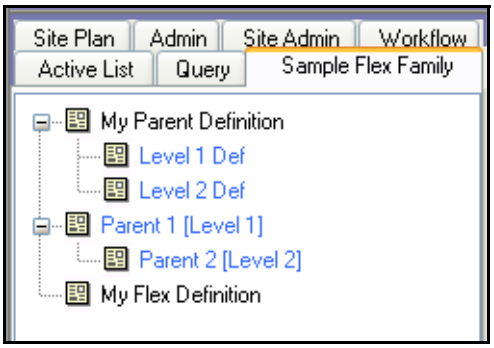
### To create the second-level parent of the hierarchy

1. From the button bar, click **New**.
2. From the list of options that appears, select **New My Parent**.

3. In the form that appears, fill in the fields as follows:

| Field Name | Enter or Select | Comments |
|---|---|---|
| Name | Parent 2 [Level 2] | This is our name for a level 2 parent in the hierarchy (in our example, the name is genericized simply to help you identify the level).<br>**Note:** At the end of this tutorial, you will change the name to a business-specific name, "Mountain Climbing" in our example (an appropriate name given that Parent 1 [Level 1] is "Sports Equipment"). |
| My Parent Definition | Level 2 Def | Selecting **Level 2 Def** places the parent you are creating at the second level of the hierarchy. |

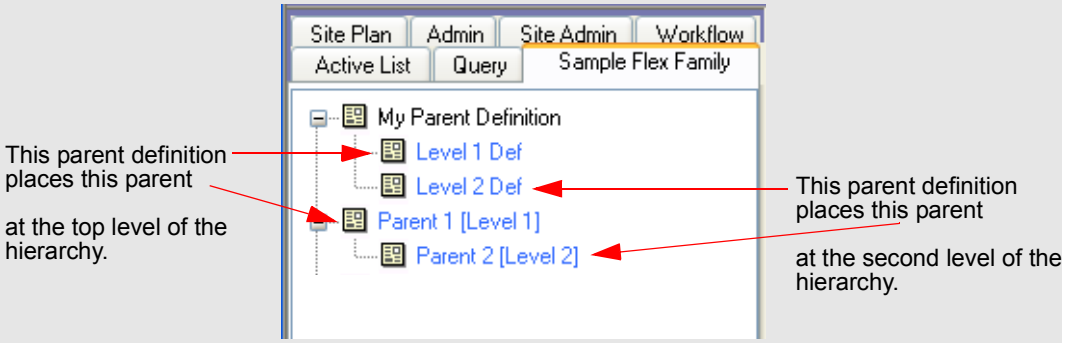
4. In the next form, click **Save**.
5. Refresh the screen.
6. Click the "Sample Flex Family" tab and expand its contents to make sure they are identical to the display below:




**Note**

Before going to the next step, review the figure on page 865. The figure summarizes how the objects that you created—parent definitions and parents based on the definitions—relate to each other.

**Parent Definitions and Parents**



## Step 7: Create Flex Definition Assets

In this step, you will create three flex definition assets:

- The first flex definition asset will be used to place assets under Parent 1 [Level 1].
- The second flex definition asset will be used to place assets under Parent 2 [Level 2].
- The third flex definition asset will be used to place the asset under both levels of the hierarchy.

**To create the first flex definition asset**

1. From the button bar, click **New**.
2. From the list of options that appears, select **New My Flex Definition**.
3. In the form that appears, fill in the fields as follows:

| Field Name | Enter or Select | Comments |
|---|---|---|
| Name | Flex Def 1 | |
| My Parent Definitions | Level 1 Def<br>**Note:** Under "Single Value," click the **Required** arrow. | Choosing **Level 1 Def** and **Single Value** means that when you use the current flex definition to create flex assets, the assets can be placed under *only one* parent that use Level 1 Def as its parent definition. In our example, the asset will be placed under Parent 1 [Level 1].<br>**Note:** Selecting a "Multiple Values" option would allow you to place the asset under *any* and *all* parents that use Level 1 Def as their parent definition. |

4. Click **Save**.

**To create the second flex definition asset**

1. From the button bar, click **New**.
2. From the list of options that appears, select **New My Flex Definition**.
3. In the form that appears, fill in the fields as follows:

| Field Name | Enter or Select | Comments |
|---|---|---|
| Name | Flex Def 2 | |
| My Parent Definitions | Level 2 Def<br>**Note**: Under "Single Value," click the **Required** arrow. | Choosing **Level 2 Def** and **Single Value** means that when you use the current flex definition to create flex assets, the assets can be placed under *only one* parent that uses Level 2 Def as its parent definition. In our example, the asset will be placed under Parent 2 [Level 2].<br>**Note:** Selecting a "Multiple Values" option would allow you to place the asset under *any* and *all* parents that use Level 2 Def as their parent definition. |

4. Click **Save**.

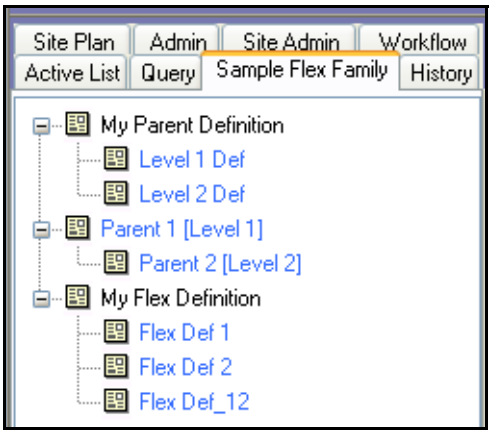
**To create the third flex definition asset**

1. From the button bar, click **New**.
2. From the list of options that appears, select **New My Flex Definition**.
3. In the form that appears, fill in the fields as follows:

| Field Name | Enter or Select | Comments |
|---|---|---|
| Name | Flex Def_12 | |
| Parent Definitions | Level 1 Def<br>Level 2 Def<br>**Note:** Under "Single Value," click the **Required** arrow. | Choosing **Level 1 Def** *and* **Level 2 Def** and **Single Value** means that when you use the current flex definition to create flex assets, the assets will be placed under *only one* parent that uses Level 1 Def *and* under *only one* parent that uses Level 2 Def as parent definitions.<br>In our example, the assets will be placed under Parent 1 [Level 1] *and* Parent 2 [Level 2]). |

| Field Name | Enter or Select | Comments |
|---|---|---|
| | | **Note:** Selecting a "Multiple Values" option would allow you to place the asset under *any* and *all* parents that use Level 1 Def and Level 2 Def as their parent definitions. |

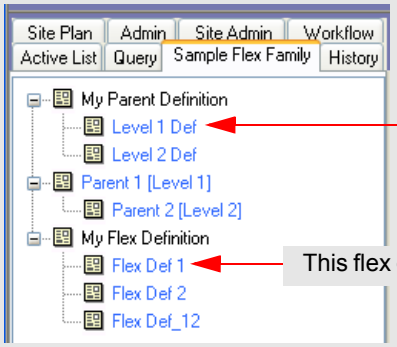
4. Click **Save**.
5. Refresh the screen.
6. Click the "Sample Flex Family" tab and expand its contents to make sure they are identical to the display below:




**Note**

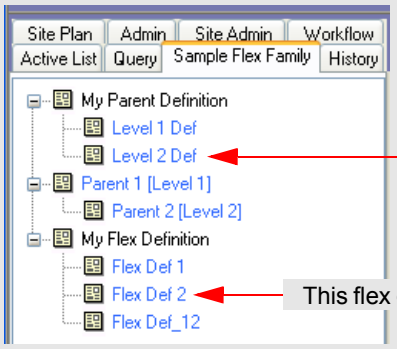
Before going to the next step, review the figure on page 869. This figure summarizes how the objects that you created—flex definitions—relate to the parent definitions they are based upon.

## Flex Definitions and Parent Definitions



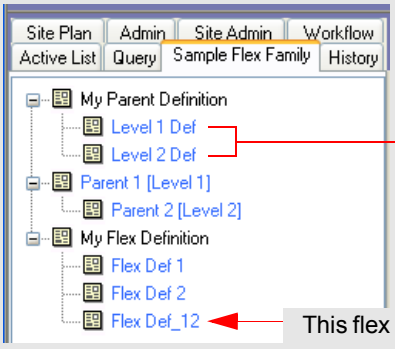

This flex definition is based on this parent definition.

(When used to create assets, this flex definition places the assets under Parent 1 [Level 1].)




This flex definition is based on this parent definition.

(When used to create assets, this flex definition places the assets under Parent 2 [Level 2].)




This flex definition is based on both these parent definitions.

(When used to create assets, this flex definition places the assets under both Parent 1 [Level 1] *and* Parent 2 [Level 2].)

## Step 8: Create Flex Assets

In this step, you will complete the flex family by adding the third level of the hierarchy—the flex assets. You will create three assets:

- The first asset you will place under Parent 1 [Level 1].
- The second asset you will place under Parent 2 [Level 2].
- The third asset you will place under both Parent 1 [Level 1] *and* Parent 2 [Level 2].

### To create the first flex asset

1. From the button bar, click **New**.
2. From the list of options that appears, select **New My Asset**.
3. In the form that appears, fill in the fields as follows:

| Field Name | Enter or Select | Comments |
|---|---|---|
| Name | Asset 1 | |
| My Flex Definition | Flex Def 1<br>**Note:** Under "Single Value," click the **Required** arrow. | Choosing Flex Def 1 will place the asset you are creating under Parent 1 (Level 1]. |

4. In the next form, click **Save**.

### To create the second flex asset

1. From the button bar, click **New**.
2. From the list of options that appears, select **New My Asset**.
3. In the form that appears, fill in the fields as follows:

| Field Name | Enter or Select | Comments |
|---|---|---|
| Name | Asset 2 | |
| My Flex Definition | Flex Def 2<br>**Note:** Under "Single Value," click the **Required** arrow. | Choosing Flex Def 2 will place the asset you are creating under Parent [Level 2]. |

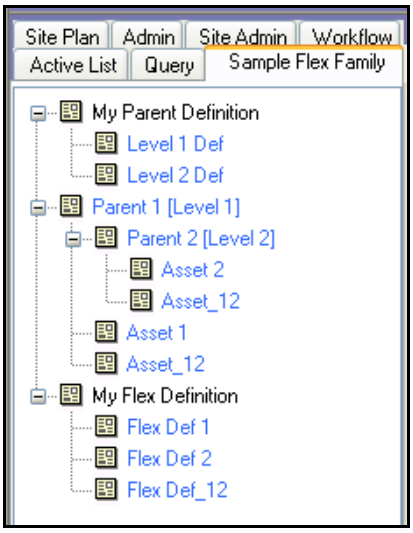
4. In the next form, click **Save**.

**To create the third flex asset**

1. From the button bar, click **New**.
2. From the list of options that appears, select **New My Asset**.
3. In the form that appears, fill in the fields as follows:

| Field Name | Enter or Select | Comments |
|---|---|---|
| Name | Asset_12 | |
| Flex Definition | Flex Def_12<br>**Note:** Under "Single Value," click the **Required** arrow. | Choosing Flex Def_12 will place the asset you are creating under both Parent [Level 1] *and* Parent [Level 2]. |

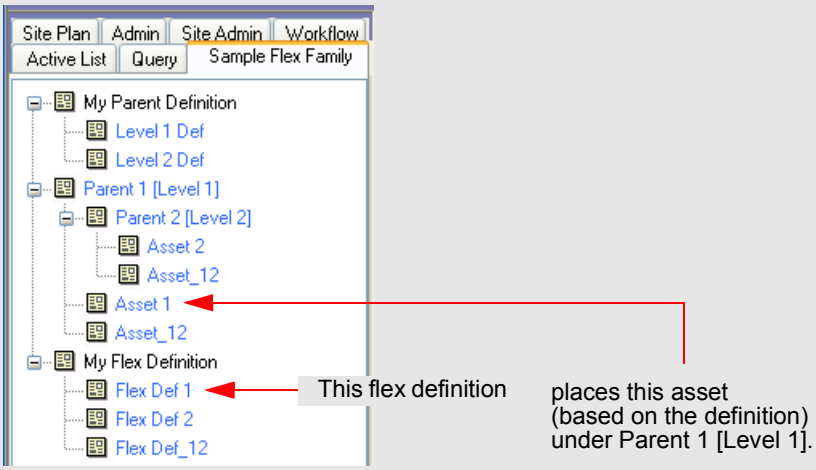
4. In the next form, click **Save**.
5. Refresh the screen.
6. Click the "Sample Flex Family" tab and expand its contents to make sure they are identical to the display below:

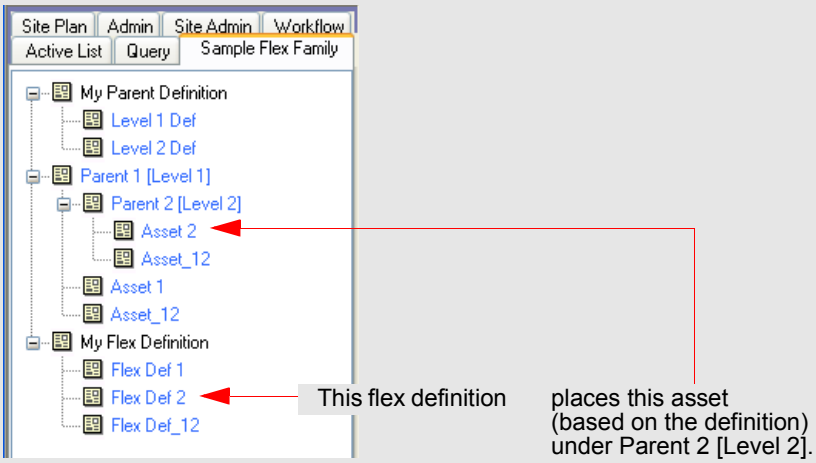


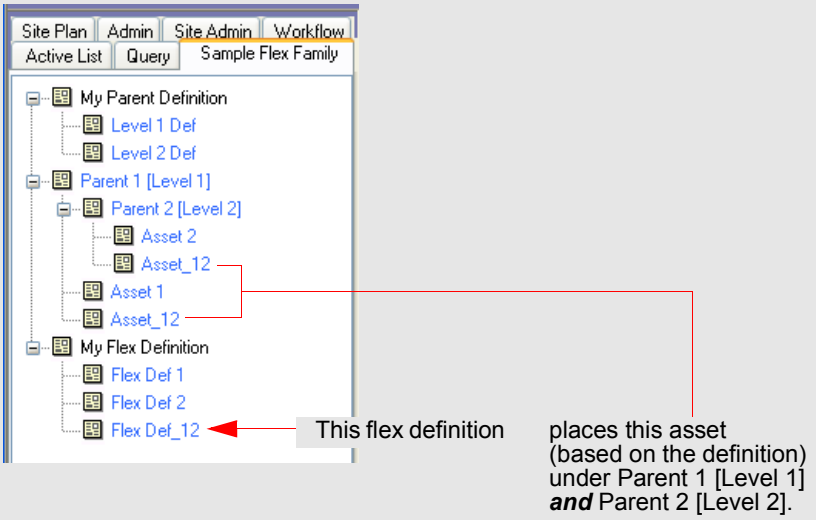

**Note**

The next figure summarizes how the objects that you created—assets—relate to the flex definitions they are based upon.

## Flex Definitions and Assets



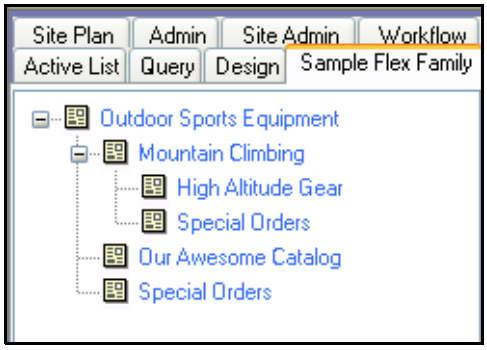



## Step 9: Translate the Formulaic Data Model into a Real-World Data Model

In this step, you will rename the parent definitions, parents, and assets in order to translate the formulaic data model you just created into a real-world model. (In practice, instead of renaming flex family members, you would name them directly in their business context, as you create them.)

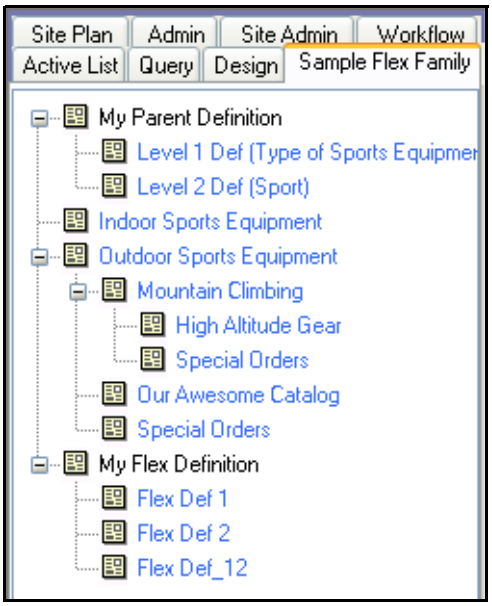
In our example, the real-world model describes a business that deals with sports gear. The flex parents and assets, after you finish renaming them, will be listed in your tree tab as shown in the figure below:




To create the real-world model:

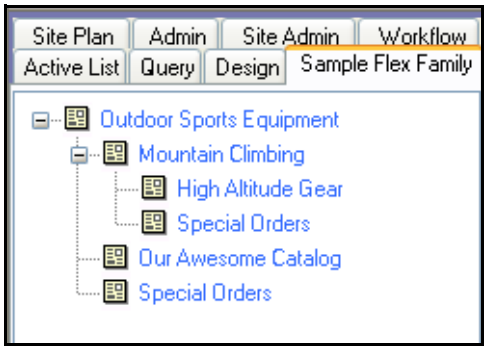
1. Rename the parent definitions as follows:
   a. In the "Sample Flex Family Tree," double-click **Level 1 Def**. Click the **Edit** button, replace the parent's name with **Level 1 Def (Type of Sports Equipment)**, and click **Save Changes**.
   b. In the same manner, replace the name of **Level 2 Def** with **Level 2 Def (Sport)**.
2. Rename the parents as follows:
   a. In the "Sample Flex Family Tree," double-click on **Parent 1 [Level 1]**. Click the **Edit** button, replace the parent's name with **Outdoor Sports Equipment**, and click **Save Changes**.
   b. In the same manner, replace the name of **Parent 2 [Level 2]** with **Mountain Climbing**.
3. Rename the assets as follows:
   a. In the "Sample Flex Family Tree," double-click on **Asset 1**. Click the **Edit** button, and replace the asset's name with **Our Awesome Catalog**.
   b. In the same manner, replace the name of **Asset 2** with **High Altitude Gear**.
   c. Replace the name of **Asset_12** with **Special Orders**.

4. Click the “Sample Flex Family” tree tab and make sure its contents are identical to the display below:




5. At this point, you can hide the parent definitions and flex definitions, and display just the parents and assets (especially for content providers who need to work with them). If you need instructions, follow the steps below:

   a. On the **Admin** tab, double-click the **Tree** node.
   b. In the “Tree Tabs” form, click **Sample Flex Family**.
   c. At the top of the form, click **Edit**.
   d. In the “Tab Contents” field, go to the “Selected” list box and Ctrl-click **My Parent Definition** and **My Flex Definition**.
   e. Click **Remove**.
   f. **Save**.
   g. Refresh the screen.
   h. Click the “Sample Flex Family” tab and make sure its contents are identical to the display below:

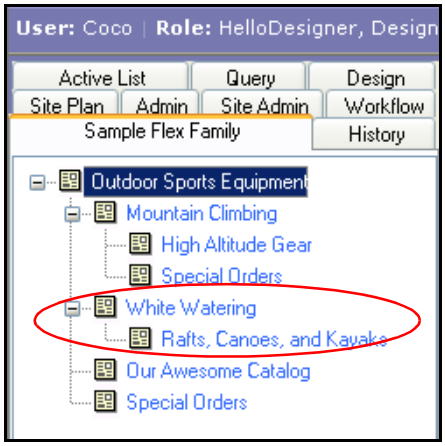


## Step 10: Develop Your Real-World Model

In this step, you will develop your data model by creating a new parent and its asset, giving each a real-world name. You will do the following:

- Create a second level-2 parent and name it **White Watering**.
- Create the parent's asset (a catalog) and name it **Rafts, Canoes, and Kayaks**.

Your display will look identical to the one below (the new members are circled).




**Note**

At the conclusion of this procedure are suggestions for further developing the real-world model, using advanced techniques. Our example illustrates the creation of *n*:1 parent-child relationships through the use of multi-valued parent and flex definitions. Guidelines, rather than step-by-step instructions are provided to help you through the development process and let you test your understanding of the process.

If you need instructions for developing your model, follow the steps below:

1. To create the level-2 parent:
   a. From the button bar, click **New**.
   b. From the list of options that appears, select **New My Parent**.
   c. In the form that appears, fill in the fields as follows:

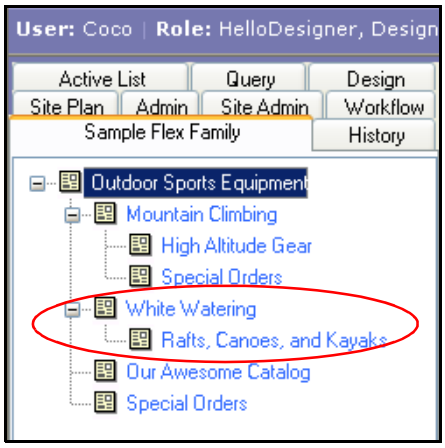
| Field Name | Value |
|---|---|
| Name | White Watering |
| My Parent Definition | Level 2 Def (Sports) |

   d. In the next form, click **Save**.

2. To create the parent's asset:
   a. From the button bar, click **New**.
   b. From the list of options that appears, select **New My Asset**.
   c. In the form that appears, fill in the fields as follows:

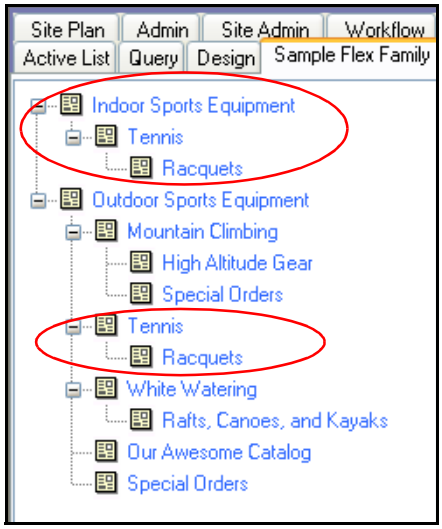
| Field Name | Value |
|---|---|
| Name | Rafts, Canoes, and Kayaks |
| My Flex Definition | Flex Def 2 |

   d. In the next form, go to the **My Parent** field and select **White Watering**.
   e. Click **Save**.
   f. Refresh the screen and display the "Sample Flex Family" tree tab. Its content should be identical to the display below (new entries are circled):



## Suggestions and Guidelines for Creating a Multi-Valued Model

Multi-valued parent and flex definitions allow *n*:1 parent-child relationships. The model at the right provides an example. As an exercise in understanding the CS implementation, we suggest that you reproduce the example. These are the steps (should you need guidelines, follow the "Guidelines" section, instead):



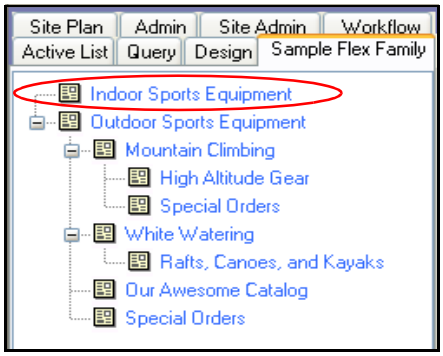

### Steps

1. Create a new level-1 parent named **Indoor Sports Equipment**.
2. Create a level-2 parent named **Tennis**. Because it is played as an indoor and an outdoor sport, you will place **Tennis** under both **Indoor Sports Equipment** and **Outdoor Sports Equipment**, as shown in the inset.
3. Create an asset named **Racquets** and place it under the **Tennis** parent, as shown.

### Guidelines

1. Create a new level-1 parent and name it **Indoor Sports Equipment**.

   For guidelines, see Step 6: Create Flex Parent Assets (page 863).

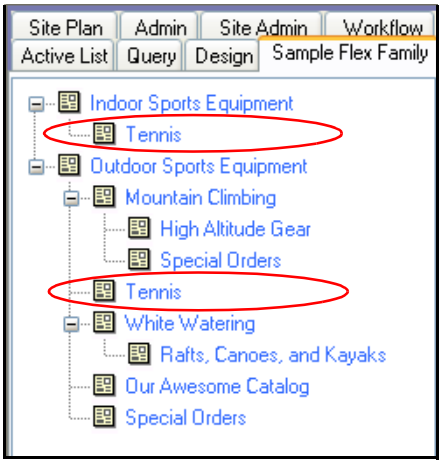



2. Create a new level-2 parent, name it **Tennis**, and place it under both level-1 parents (**Indoor Sports** and **Outdoor Sports**).

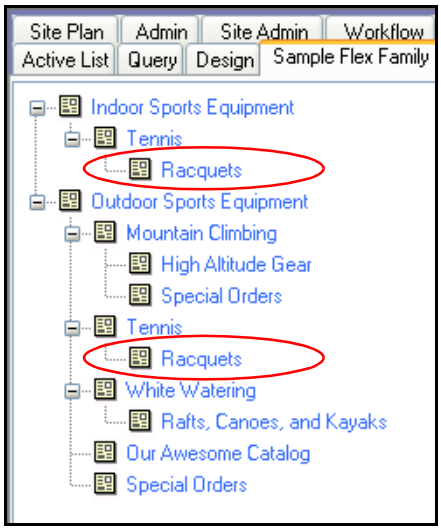
   The guidelines are:

   a. Change **Level Def 2 (Sport)** to have an optional and multi-valued level 1. (This re-definition makes it possible to place a level-2 parent under any and all level-1 parents). For reference, use the procedure in Step 5: Create Parent Definition Assets (page 861).

   b. Create the level-2 parent (**Tennis**). For reference, use the procedure in Step 6: Create Flex Parent Assets (page 863). In the **My Parent** field, make sure to select both options (**Indoor Sports Equipment** and **Outdoor Sports Equipment**).



3. Create a new asset named **Racquets** and place it under the **Tennis** parent (both entries).




   The guidelines are:

   a. Change **Flex Def_12** to have optional and multi-valued parent definitions. (This re-definition makes it possible to place an asset under any and all level-1 and level-2 parents.) For reference, use the procedure in Step 7: Create Flex Definition Assets (page 866).

   b. Create the asset (**Racquets**). For reference, use the procedure in Step 8: Create Flex Assets (page 870). When populating the asset form, go to the **My Parent** field and select **Tennis** for Level 2 Def (Sport).

# Next Steps

At this point you have created a rudimentary data model. From here, you can expand the data model by creating additional parents to occupy a given level, creating additional levels (by chaining flex parent definitions), and so on. You can also create associations among assets, write template code to format the assets, and test the publishing options that display the assets to site visitors. For information on these operations, see Chapter 16, “Designing Flex Asset Types.”

Appendix B

# Content Server URL Assemblers

This appendix explains Content Server URL assemblers, which manage URL assembly and disassembly, and provide an interface that you can use to define the appearance of URLs.

This appendix contains the following sections:


- Overview of Content Server URL Assemblers
- Assemblers Installed with Content Server
- Working with Assemblers

# Overview of Content Server URL Assemblers

URL assemblers, in conjunction with URL generation tags, are used to generate Content Server URLs and to disassemble the URLs they generate.

## URL Assembly

Content Server URL generation tags (`<satellite.link>`, `<satellite.blob>`, `<render.getpageurl>`, `<render.getbloburl>`, `<render.satelliteblob>`, `<rendergettemplateurl>`) are used to construct a link to a Content Server resource, such as a page or a blob. The data, such as tag attributes and nested argument tags which you specify when using the tag, is converted into an abstract object called a **URL definition**. The URL definition is passed into the **URL assembler**. The URL assembler then converts the definition into a string URL that is returned.

Two assemblers are installed with Content Server, but you have the option of creating your own assemblers in order to directly control the appearance of your URLs. Before you can use the assemblers you create, you must first register them with Content Server.

The assembler that is configured as the default is used to create all Content Server URLs. You can change the default assembler. You can also override the use of this default assembler in individual link tags.

## Assembler Discovery and Disassembly

Because an assembler can create a URL in any form that the assembler's author dictates, it may be impossible for the URL to be decoded into parameters by an application server when an assembled link is requested. For decoding to take place, the assembler must be able to disassemble the string URL into its definition. Assemblers are therefore reversible, that is, capable of disassembling any URLs that they assembled.

If a URL has been created using an assembler other than the default assembler, then the default assembler cannot disassemble the URL. At that point, the next highest ranked assembler attempts to disassemble the URL. If it succeeds in creating a definition, then the assembler engine is said to have “discovered its assembler,” and the definition is converted into parameters for processing. If the next highest ranked assembler fails to disassemble the URL, the third highest ranked assembler is called upon to disassemble it. This process continues until the URL is successfully disassembled. Note that this process requires an assembler to be able to recognize the URLs it assembled as its own, and all other URLs as foreign.

See “Creating Assemblers” on page 882 and “Registering and Ranking Assemblers” on page 882 for more information about creating, registering, and ranking assemblers.

## URL Assembly and Disassembly Using GET and POST Requests

URL assemblers are only invoked on `GET` requests. They are not invoked on `POST` requests. For example, when accessing a page with a `GET` request, the URL assembler is invoked to disassemble the URL. It then provides the appropriate parameters that Content Server requires to open that page (such as `c`, `cid`, and `pagename`) by adding them to the definition (if they do not already exist in the definition). However, when a request is POSTed, such as a form with `method=post`, the URL assembler is not invoked to disassemble the URL. Therefore, the parameters Content Server requires to open the page must be part of the post request itself.

This can be accomplished by encoding the following tag into the page's template or element (replacing the sample values with the parameters Content Server requires for the page's URL):

```
<satellite:form method="post" id="assetid">
<render:gettemplateurlparameters list="args" ... /><!-- add all
parameters that are normally part of a URL -->
        <ics:listloop listname="args">
               <input type="hidden" name="<string:stream
list="args" column="name"/>" value="<string:stream list="args"
column="value"/>"/>
        </ics:listloop>
```

# Assemblers Installed with Content Server

The two assemblers that are installed with Content Server are Query Assembler and QueryAsPathInfo Assembler.

## Query Assembler

The Query Assembler creates URLs with query strings. It is the default assembler, and it is automatically registered in Content Server. Therefore, until you make any modifications (such as changing the default assembler or overriding the default in link tags), Query Assembler will be used to generate all URLs.

## QueryAsPathInfo Assembler

The QueryAsPathInfo Assembler does not use query strings. Instead, the QueryAsPathInfo Assembler encodes the query string and appends it to the end of the servlet name. The benefit of this assembler is that it creates URLs that can be indexed by search engines. The QueryAsPathInfo Assembler is not automatically registered with Content Server.

**Note**

A sample URL assembler implementation ships with the FirstSite II sample site. The source code is located on the Content Server CD:

```
/ContentServer/FirstSiteII/PrettyURL/src/com/
   fatwire/firstsite/uri/FSIIAssembler.java
```

Additional information is available in Javadocs. The classes of interest are:

- `Definiton`
- `Definiton.AppType`
- `Definiton.ContainerType`
- `AbstractAssembler`
- `AbstractAssembler.AssemblyContext`
- `AbstractAssembler.DisAssemblyContext`
- `Assembler`
- `DisAssembler`

It may also be beneficial to set up a debug environment with `FSIIAssembler.java` so that you can step through the code.

# Working with Assemblers

This section explains how to create and register your own assemblers. This section also explains how to modify link tags to override the use of the default assembler.

## Creating Assemblers

The Content Server URL Assembly module enables you to create your own assemblers. This option gives you direct control of the appearance of your URLs.

### To create an assembler

1. Write a java class that implements the `com.fatwire.cs.core.uri.Assembler` interface. For information about this class, see the *JavaDocs*, which are located at the following URL:

   `http://e-docs.fatwire.com`
2. Compile the class into a `.jar` file.
3. Deploy your class into the Content Server web application and the web application for each remote Satellite Server you have installed.

   This usually means copying the `.jar` file you just created into your web application's `WEB-INF/lib` folder. For remote Satellite Servers, this means copying it to the `resin/webapp/ROOT/WEB-INF/lib` folder.
4. Register your new assembler in the `ServletRequest.properties` file on both Content Server and all of your remote Satellite Servers (see "Registering and Ranking Assemblers" on page 882 if you need instructions).
5. Restart Content Server and all of your remote Satellite Servers.

## Registering and Ranking Assemblers

Before an assembler can be used to create URLs, it must first be registered with Content Server. The registration is done by listing assembler class names with corresponding short forms in a property file. The registration also includes a ranking that indicates in which order the assemblers should be used.

### To register an assembler

1. Invoke the Property Editor.
2. Open the `ServletRequest.properties` file.
3. Click the **URI Assembler** tab to access the assembler properties.
4. Specify the `classname` and `shortform` of the assembler you want to register.

   The third element in the property name indicates the ranking of the assembler. The assembler with the ranking of `0` is the highest ranked (and default) assembler, the assembler with the ranking of `1` is the next highest ranked, and so on.

   If you want to configure the new assembler to be the default assembler, enter the `classname` and `shortform` values in the properties that have `1` as their ranking.

For example, the syntax to register the QueryAsPathInfo assembler as the default assembler would be as follows:

| Property Name | Property Value |
|---|---|
| `uri.assembler.0.classname` | `com.fatwire.cs.core.uri.QueryAsPathInfoAssembler` |
| `uri.assembler.0.shortform` | `pathinfo` |
| `uri.assembler.1.classname` | `com.fatwire.cs.core.uri.QueryAssembler` |
| `uri.assembler.1.shortform` | `query` |

**Note**

Make sure that the Query Assembler is always registered, even if you have lowered its ranking, because it is used within CS-Direct. The Query Assembler must be registered with the `shortform` value of **`query`**.

5. Depending on the ranking of the new assembler, you may need to adjust the rankings of the other assemblers. Verify that all of the assemblers are configured and ranked correctly in the property file. If they are not, make any necessary changes.
6. Choose **File > Save** to save your changes and close the Property Editor.
7. Repeat steps 1–6 for each remote Satellite Server you have installed.
8. Restart Content Server and all of your remote Satellite Servers.

## Modifying Link Tags

Content Server link tags can be modified to use an assembler other than the default assembler. The link tags accept an attribute, `assembler`, which take an assembler short form as a value.

For example, to override the default assembler with the QueryAsPathInfo assembler in an individual link tag, the syntax would be as follows:

```
<satellite:link pagename="example" assembler="pathinfo" />
```

Appendix C

# White Space and Compression

When Content Server streams a text page, the page may contain a significant amount of "white space"—spaces, carriage returns and tabs—that have no effect on the data that is consumed by the client. The white space is visible when the source code is viewed by the consumer, and this is not desirable. Furthermore, excessive white space needlessly increases the size of the response, which ultimately increases bandwidth use. Consequently, it is beneficial to eliminate white space whenever possible.

This appendix contains the following sections:


- White Space and JSP
- White Space and XML
- Compression
- JSP Design

# White Space and JSP

The JSP specification requires that all white space be preserved. Thus, a page that looks like this:

```
<%@ page import="my class name"%>
<%@ page import="my class 2"%>
<cs:ftcs>
<p>Hello world!</p>
</cs:ftcs>
```

will have three carriage returns and a tab preceding the `<p>` because the text is displayed on third line after the JSP has been interpreted. With more complicated pages, the problem is compounded.

# White Space and XML

Content Server's XML processing language, being a proprietary set of xml-compliant tags, does not adhere to the white space preserving rules of JSP. As such, a CS XML page like this:

```
<? XML version 1.0 ?>
<FTCS>
<p>Hello World!</p>
</FTCS>
```

will display `<p>` as the first characters of output, because our xml parser will strip all of the white space (unless xml debug is enabled, in which case all the white space is preserved).

# Compression

Because white space is an artifact of writing well-formatted code, its presence is an unfortunate side effect of programming practices that benefit the developer. The impact on the consumer and the customer is minimal except for bandwidth. To address bandwidth, the output of all text-based pages can be compressed. Compressing the output is done on the server side, and decompression is done by the consumer's user-agent (browser). The compression/decompression is completely transparent to the end user. This sort of compression can yield up to an 80% reduction in bandwidth use. One commonly-used compression mechanism is the mod-gzip extension to the Apache web server. This module will automatically gzip all output to the user agent provided that it can decompress it. Configuration is minimal and its effectiveness is quite high. It can be obtained from SourceForge (`http://sourceforge.net/projects/mod-gzip/`). Similar tools are available for other common web servers such as IIS.

Another possibility is to do the compression at the application server layer, and leave the web server alone. This is best done by connecting a standard servlet filter to Satellite Server (or to Content Server if Satellite Server is not being used). The servlet filter is invoked in a prescribed order prior to and/or after the invocation of the specified servlet, and during invocation it can compress the output prior to sending it to compatible user-agents, exactly the same way mod-gzip works. One such compression filter can be found at SourceForge (`http://sourceforge.net/projects/pjl-comp-filter/`).

If you are interested in compression but need assistance, contact FatWire Professional Services.

# JSP Design

If compression is not an option, consider altering your JSP pages so that they do not require compression to address the white space problem. This can be done by changing the code above to this:

```
<%@ page import="my class name"
%><%@ page import="my class 2"
%><cs:ftcs><p>Hello world!</p></cs:ftcs>
```

While this is not as elegant (or readable), it will result in page output without any white space whatsoever prior to the `<p>` tag. An intermediate solution may be something like this:

```
<%@ page import="my class name"
%><%@ page import="my class 2"
%><cs:ftcs>
<p>Hello world!</p>
</cs:ftcs>
```

For extensive examples of how to address white space issues in JSP, refer to our WebServices elements in the ElementCatalog. They are included with CS Direct.

Appendix D

# Asset and Publish Events in Content Server

This chapter takes you through the process of creating event listeners.


- Overview
- Types of Events
- Asset Events
- Publishing Events

# Overview

Content Server's event framework enables developers to write custom business logic to execute upon certain specific events in Content Server. The custom business logic needs to be implemented in the form of Java classes in the classpath that implements a pre-defined interface. This tutorial takes you through the process of creating listeners.

# Types of Events

Two types of events are currently supported by Content Server: Asset events and Publishing events.

- Asset events are the events that happen when assets are added, modified or deleted, either by a user or programmatically. Upon these events, the event framework looks up and executes the set of configured events.
- Publishing events are events that the RealTime publishing framework generates at each step of the publishing process. This feature is intended to facilitate system monitoring (such as SNMP) and other housekeeping processes.

# Asset Events


- Writing an Asset Event Listener
- Registering an Asset Event Listener


## Writing an Asset Event Listener

Asset listeners need to extend `AssetEventListener` in order to be notified of asset changes. A convenient base class `AbstractAssetEventListener` comes with Content Server. Extending from this class makes it easy to recognize the specific type of action that led to the event (add/modify/delete).

The following sample code implements a custom asset listener. All it does is print the asset IDs; however, it is easy to see how one could plug in custom business logic.

```
package com.mycompany
public final class CustomAssetEventListener extends
AbstractAssetEventListener
{
    public void assetAdded(AssetId id)
    {
        System.out.println("Asset " + id + " added");
    }
    public void assetUpdated(AssetId id)
    {
        System.out.println("Asset " + id + " added");
    }
    public void assetDeleted(AssetId id)
    {
        System.out.println("Asset " + id + " added");
    }
}
```

Asset event listeners are invoked after the asset operation has taken place, but before committing the data.

## Registering an Asset Event Listener

**Caution!**

Content Server ships with a standard listener that is used for search indexing. Do not alter or delete it.

Asset event listeners are registered in the `AssetListener_reg` database table. Here is the structure of the table:

| ID (integer) | Unique Identifier for the Row |
|---|---|
| `listener(String)` | Fully qualified class name that implements `AssetEventListener`. For example: `com.mycompany.CustomAssetEventListener` |

| ID (integer) | Unique Identifier for the Row |
|---|---|
| `blocking(Y or N)` | ‘`Y`’ indicates that the listener is blocking (runs synchronously with the thread that generated the event).<br>‘`N`’ indicates that the listener is non blocking (runs in a separate thread). |

# Publishing Events

- Writing a Publishing Event Listener
- Registering a Publishing Event Listener

## Writing a Publishing Event Listener

Publishing listeners need to implement `PublishingEventListener`. A `PublishingEvent` passed into the listener indicates the specific event that caused the invocation.

The following example shows a custom implementation that prints the `pubsession` ID to the console.

```
package com.mycompany;
public class CustomPublishingEventListener implements
PublishingEventListener
{
    public void onEvent( PublishingEvent event ) throws EventException
    {
        System.out.println( "Publishing event fired for    pubsession:
" + event.getPubSessionId());
        System.out.println( "Publishing task : " +
event.getTaskName());
        System.out.println( "Status of the task :  " +
event.getStatus());
        System.out.println( "Message associated with the task : " +
event.getMessage() );
    }
}
```

Publishing consists of multiple tasks (data gathering, packaging, transport, and so on), and each of them generates events. The `PublishingEvent` class represents an event in the publishing task. An implementation can query the task and its status from the event as shown above.

Each task generates events when the following states are reached:

- STARTED
- DONE
- CANCELLED
- SUBTASK_FINISHED
- FAILED

## Registering a Publishing Event Listener

Publish event listeners are registered in the `FW_PublishingEventRegistry` database table. Here is the structure of the table:

| ID (integer) | Unique Identifier for the Row |
|---|---|
| `listener(String)` | Fully qualified class name that implements `AssetEventListener`. For example `com.mycompany.CustomPublishingEventListener` |
| `blocking(Y or N)` | ‘`Y`’ indicates that the listener is blocking (runs synchronously with the thread that generated the event).<br>‘`N`’ indicates that the listener is non blocking (runs in a separate thread). |

# Index

## A

access ID 800

accounts
  coding account management forms 692

ACL (access control list)
  and caching 119
  and live site security 688
  and site visitors 688
  identifying a visitor's ACL 690
  SiteGod 692
  SystemACL table 215

ACL tag family 688

Add Enumerated Value field 782

addData element 431

adding
  mimetypes 300
  rows to tables 234

ADF, *See* asset descriptor files

aliases 795

AMap tables 203

APIs
  Directory Services 682

AppendSelectDetails element 275

approval dependencies 580
  ASSET.LOAD 586
  ASSETSET tag family 587
  collection assets 594
  exact 582
  exists 581
  none 582
  publishing 580
  RENDER.FILTER 589
  RENDER.GETPAGEURL 587
  RENDER.LOGDEP 588

approval templates 583
  designating 537

arguments
  in XMLPost configuration files 413

article assets 186
  coding templates for 591

asset descriptor files 186
  and embedded links 285
  basic format of 268
  coding 280
  fine-tuning 291
  uploading 288

asset types
  attribute editor 184, 200
  collection 183
  core 183
  creating basic asset types 278
  creating flex asset types 312
  CSElement 183
  defined 182
  defining columns for basic asset types 270
  defining their database columns 270
  deleting basic asset types 302
  enabling fields for 285
  naming 269

page 183
query 183
registering elements for 290
sample site 186
SiteEntry 183
template 183

ASSET.CHILDREN 594

assetid 798

AssetMaker
and embedded links 285
creating basic asset types 186, 264
in Burlington Financial 719

AssetPublication table 561

AssetPublicationTree table 190

AssetRelationTree table 187, 190, 213

assets 99
*See also* template, query, collection, imagefile, stylesheet, page, flex attribute, flex asset, flex definition, flex filter, flex parent, and flex parent definition
adding mimetypes for 300
and tags 207
basic 45, 182
child 621
CS-Direct Advantage 196
database tables for basic assets 190
database tables for flex assets 203
defined 182
deleting 556
designing 181
differences between basic and flex 208
Engage 184
flex 54, 182
flex family 196
flex filter 196
getting field values 99
held 580
importing 430
indexing 207
listing children 99
loading 98
loading basic 98
retrieving field values 99
sharing 555
summary of basic and flex 208
which model to use 185

assetsets 206
and flex attribute asset types 601
and searchstates 600
attributes of type blob 606
attributes of type text 605
attributes of type url 608
creating 603
getting attribute values 604
list objects 609
overview 57, 599

AssetStubCatalog table 265

assettype column 798

AssetType table 264

Associated queries field (collection assets) 558

Association
form 187
table 187

associations 187
creating fields for 296
deleting association fields 298
flex assets 328

attribute editors 184, 200
checkbox 356
code syntax for 355
components of 352
creating 365
custom 367
editing 372
elements 363
elements for 361
eWebEditPro 357
pick asset 358
presentationobject.dtd 352
pull-down 358
radio buttons 359
textarea 360
textfield 361
XMLPost configuration file 435
XMLPost source file 435

attributes 60, 196
categories 779
creating 307
deciding how many types 307
flex filter assets 307
foreign 320
history 184, 783
inheritance of values 200
of type asset 308

visitor 184, 780

*authpassword* variable 105

*authusername* variable 105

## B

basic asset model
advantages 185
data structure 190
overview 186
parent-child relationships 187

basic assets
and embedded links 285

BIGINT data type 218, 271

binary visitor attributes 780, 797
and storing carts across sessions 800
collecting visitor data 795
retrieving 801

blob (binary large object) 614
BlobServer URLs 606
caching 123
code example for displaying 623
coding elements for 593
creating URLs for 95
displaying value of 607
flex attributes of type blob 606

BLOB data type 271

BlobServer 593
BLOBSERVICE tags 607
servlet 36
URL for attribute of type blob 606
URLs 614

BulkApprover
configuring 486
overview 486
using 488

BulkLoader
and DB2 465
command line 475
configuration file 468
configuration file sample 473
custom extraction mechanism 476
description 462
flat tables 464
IDataExtract 477
IFeedback 485
input table 465
IPopulateDataSlice 481
java interfaces 476
mapping table 467

Burlington Financial sample site
AssetMaker assets 719
bfmembers table 695
caching 722
collection assets 720
database searching 715
email to a friend page 634
home page 620
mimetypes 719
navigation elements 713
navigational bar 636
overview 51, 58, 712
plain text link 627
query assets 715, 717
section pages 625
site plan 67
today's date function 639
visitor authentication 695, 720
Wirefeed query 631

## C

*c* variable 105

cache
and pagecriteria 126
CacheManager 126
the cache key 126
viewing the contents of 125

Cache Criteria 535

cache criteria 514, 535

cache key
overview 126

cached elements
calling 94

cacheinfo column 722

CacheManager
and mirror publishing 585
and preview 585
Satellite Server 584

caching
cache key 126
compositional dependencies 584
disable individual pages 617
flushing resultsets 259

load on database 257
overview 40
properties 260
resultsets 42, 256

calculating
approval dependencies 580
promotions 797
segments 797

calling
CSElement asset 496, 589
element 496
page name 496
SiteEntry asset 496, 589
uncached elements 94

CALLSQL
example of 258

CatalogManager
servlet 36, 230
tag attributes 230

CatalogMover
command line interface 160
connecting to Content Server 154
exporting database tables 156
exporting to ZIP files 157
importing database tables 158
menu commands 154
overview 153
starting 153

catalogs
*See also* database tables

categories 779, 781, 787
adding 298

category
column 188, 194
field 560

Category field
collection asset 558
page asset 567
query asset 563
stylesheet asset 565

Category table 214, 264

cc.contentkey property 214

CGI
programming 107

CHAR data type 217, 271

characters, special 91

checkbox
attribute editor for flex attributes 356
input type for basic asset fields 273

child assets
basic assets 187

child nodes 637

child templates
example 627

*cid* variable 105

codes, status 560

coding
account creation form 691
asset descriptor files 280
attribute editor assets 355
error tracking 177, 178, 616
login forms 689
source files for XMLPost 421

coding elements
error tracking 177
for basic assets 591
for blobs 593
for collections 594
for flex assets 599
for imagefile assets 593
for page assets 597
for query assets 595
for recommended assets 799
logging dependencies 585
shopping cart 800
that collect visitor data 796
workflow elements 747

collecting visitor data 60, 795
coding pages 796
debugging 801

COLLECTION argument 798

collection assets 183
and query assets 556, 561
approval dependencies 594
coding elements for 594
compositional dependencies 595
creating 557
definition 556
example code for creating links 625
example element code 621
in Burlington Financial 720

sharing 558

column constraint types 218

columns
defining for basic asset types 270

commerce context 794
defined 794
tags 794

commerce ID 800

compositional dependencies 580
and page generation 584
ASSET.LOAD 586
ASSETSET tag family 586
CacheManager 584
collection assets 595
query assets 596
RENDER.LOG 587
RENDER.UNKNOWNDEPS 589

conditionals 97, 111

configuration files
XMLPost 410
XMLPost and flex asset types 433

connecting to Content Server 152

Constraint type field 782, 785

constraints
adding to searchstates 611
deleting from searchstate 612
range 611

Content Server
database overview 41
session variables maintained by 166
SOAP interface 764

Content Server context
creating with the ftcs tag 83
ICS object 82
overview 82

Content Server Desktop
overview 59

Content Server Direct
overview 44

Content Server Direct Advantage
asset types 196
overview 54

Content Server Explorer 152
creating variables in 106
overview 152
registering a foreign table 225
when to use 252

Content Server Management Tools
creating content tables 223
creating object tables 220
when to use 252

content tables 214
creating 223
managing the data in 225
working with 230

ContentDetails element 275

ContentEditor ACL 692

ContentForm element 275

ContentServer servlet 36

*context* variable 105

cookies 169, 795
attributes 170
example 170
removing 169

CookieServer servlet 36

core asset tags
ASSET.CHILDREN 99
ASSET.GET 99
ASSET.LOAD 98
ASSET.SCATTER 99

counter variables 114

createdby column 192

createddate column 192

creating 95
assetsets 603
association fields 296
attribute editors 365
basic asset types 278
BlobServer URL embedded in HTML tag 614
BlobServer URLs 95, 606, 614
categories 298
collection assets 557
content tables 223
CSElement assets 521, 531
entries for attribute drop-down lists 782, 786
flex asset types 312

flex attributes 307
flex attributes of type asset 308
flex definition assets 324
flex family 312
flex parent assets 326
history attributes 783
history definitions 786
hyperlinks to assets in a collection 625
object tables 220
page assets 566
parent definition assets 323
product sets 332
searchstates 603
sources 299
stylesheet assets 564
template assets 501
tree tables 222
URLs for hyperlinks 95, 613
user accounts programmatically 692
visitor attributes 780

CS
.dtd file 90
creating the Content Server context 83
JSP actions 86
JSP declarations 87
JSP directives 87
JSP expressions 87
JSP scriptlets 87
JSP syntax 86
JSP tag libraries 87
the Content Server context 82
using directives in CS 87
XML and the FTCS tag 90
XML overview 90
XML standard beginning 90
XML version and encoding 90

CS tags
for basic and flex assets 93

cs.timeout 166

CSElement assets 183, 493
as root element for SiteEntry asset 531
calling 589
creating 521, 531
deleting 538
editing 537
entry in ElementCatalog table 529
how to invoke one from an element 496
name to use when calling 526
previewing 539
sharing 538
where used 499

*ct* variable 105
used in page names 629

*currentAcl* session variable 166

*currentACL* variable 105

*currentUser* session variable 166

custom extraction mechanism, BulkLoader 476

customizing
attribute editors 367

## D

data 769
adding rows 234
coding data entry forms 234
deleting rows 237
design of 181
listing 232
querying for 232
retrieving from tables that do not hold assets 230
writing to tables that do not hold assets 230

data types 784
BIGINT 271
BLOB 271
CHAR 271
Content Server 217
CS-Direct (AssetMaker) 271
CS-Direct Advantage (flex attributes) 199
DOUBLE 271
INTEGER 271
LONGVARCHAR 271
SMALLINT 271
specifying for attribute 781
TIMESTAMP 271
valid input for 780
VARCHAR 271
web services 768, 770

database tables
AMap tables 203
AssetPublication 561
AssetRelationTree 187
bfmembers 695
content tables 214
default columns for basic asset types 191
defined 212
deleting 253

editing rows 240
exporting 156
for basic assets 186, 190
for flex assets 203
foreign tables 214
identifying the type 216
importing 158
Mimetype 300
Mungo 606
Mungo tables 203, 204
MungoBlobs 606
object tables 212
overview 41
retrieving data from non-asset tables 230
SiteCatalog 126
SitePlanTree 66
StatusCode 560
system tables 215, 226
SystemInfo 41
tree tables 213
types of 212
visitor data 795
VMVISITORALIAS 794
VMVISITORSCALARBLOB 795
VMVISITORSCALARVALUE 795
VMz 795
working with tables that do not hold assets 230

DATETIME data type 217

DB2
and BulkLoader 465

debugging 175
promotions 801
recommendations 801
session links 800
visitor data collection 801

DebugServer servlet 36

Default Arguments for Preview field 540

default storage directory, *see* defdir

default template 627

Default Value field 781

DefaultReader username 166

defdir 289
column 219
defined 219
specifying for new basic asset types 282

definitions, *See* flex parent definitions or flex definitions

deleteAsset element 459

deleting
assets 556
assets with XMLPost 459
association fields 298
basic asset types 302
constraint from a searchstate 612
CSElement assets 538
history attributes 783
page assets 572
rows from tables 237
SiteEntry assets 538
template assets 538

delivery system 35
user management on 682

dependencies
*See also* approval dependencies and compositional dependencies
approval 580
code that logs 585
compositional 580

dependency log 580

description column 191

Description field
page assets 558, 563, 564, 567
templates 507

descriptions
visitor data assets 780

development process
recommendations 790
visitor data assets 779

development system 35

DIR tag family 683

directory
entry 682
groups 682
hierarchies 682
operations 684

directory operations
code samples 684
error handling 686
troubleshooting 687

Directory Services API 682

discounts 61

DOUBLE data type 218, 271

drop-down fields
  examples 286
  history attributes 786

dynamic publishing
  Mirror to Server 53, 580
  overview 39

## E

ED status code 560

editing
  attribute editors 372
  CSElement assets 537
  flex assets with XMLPost 457
  history attributes 783
  rows in tables 240
  SiteEntry assets 537
  template assets 537

*eid* variable 105

ElementCatalog table 215

elements 594
  beginning 90
  coding 579
  editing search elements for basic asset types 301
  ending 90
  examples 620
  ExecuteQuery 559, 596, 633
  for attribute editors 361
  for flex asset types 311
  no cache criteria 495
  PostUpdate 292
  PreUpdate 292
  registering with AssetMaker 264
  required tags 90
  that work with attribute editors 352
  when use non-asset elements 500

embedded links
  and basic assets 285
  and flex assets 360
  displaying 740

enabling
  standard fields for basic asset types 285

encoding
  XML 90

encryption 689

end date 561

enddate
  column 193
  field 285

Engage
  asset types 184

entities 91

entry
  directory 682

enumerated lists 782, 786

*errdetails* variable 177

*errno* variable 105, 177, 178, 616

error logging 175, 616

error tracking 177, 178, 616

events
  SystemEvent table 215

eWebEditPro
  attribute editor for flex attributes 357
  configuring 404
  input type for basic asset fields 273

exact dependency 582

ExecuteQuery element 559, 596, 633

exists dependency 581

expiration
  of cookies 170
  of sessions 166

Export to Disk publishing 53, 583
  approval templates 583

exporting
  database tables 156
  to ZIP files 157

externaldoctype column 195

## F

field (column) types
  database-specific 218

filename
  column 194
  field 285

Filename field
  page assets 567

Filter by field 784

firewalls
  and XMLPost 411

flex asset model
  parent-child relationships 197
  when to use 185

flex assets
  and embedded links 360
  assetset examples 799
  associations 328
  designing 306
  displaying attribute values of 602
  flex asset types 55, 196
  flex definitions 202
  importing 430
  importing with BulkLoader 464
  searchstate examples 600
  tag families to use 599

flex attributes 196
  blob type and assetsets 606
  checkbox attribute editor 356
  data types 199
  displaying their values 602
  eWebEditPro attribute editor 357
  example XMLPost configuration file 438
  example XMLPost source file 439
  extracting values of with assetsets 604
  foreign attributes 320
  importing 436
  inheritance of values 200
  pick asset attribute editor 358
  pull-down attribute editor 358
  radio button attribute editors 359
  text attributes and assetsets 605
  text field attribute editor 361
  textarea attribute editor 360
  types of and assetsets 601
  url type and assetsets 608

flex definitions 196
  approval templates 584
  as business rules 201
  assigning flex filter assets to 326
  create hierarchy on tree tabs 200
  creating 324
  deciding how many types 309
  example XMLPost source file 443
  importing 440

flex families 196
  creating 312

flex filter assets 196
  assigning to flex definition assets 326
  assigning to flex parent definition assets 324

flex filter classes
  defined 335

flex parent definitions 196, 200
  assigning flex filter assets to 324
  setting hierarchical place 310
  specifying attribute inheritance 310

flex parents 196, 200
  creating 326
  specifying for flex asset when importing 453

FlushServer servlet
  overview 44

foreign attributes 320

foreign tables 214
  managing data in 226
  registering 225

forms 107

*ft_ss* variable 105

*ftcmd* variable 104

full-text search 57, 206, 601

futuretense.ini file 161

futuretense.txt 176

## G

GAProductSet attribute 332

GE Lighting sample site
  content asset types 198
  overview 58
  product asset types 197

generic field (column) types 217

getting
  the children of a basic asset 594
  values from asset fields 99

groups
directory 682

## H

held assets 580

Hello Asset World sample site 697

HelloCS
servlet 37

hierarchy
directory 682

history attributes 60, 184
creating 783

history definitions 60, 184
creating 786
example code 796
retrieving data stored as 801

HTML tags
including an XML variable in 624, 627, 638
substituting XML variables in 110
with BlobServer URL embedded in 614

hyperlinks 613
creating 95

## I

ICS Object
overview 82

ics.disable cache 617

id column 191, 204

IDataExtract interface 477

identifying
a foreign table 225
type of database table 216
visitors 794

IDs
access 800
visitor 794

IFeedback interface 485

IList object
tags 232

ILists
web services 769

image assets 186
coding templates for 591

IMAGE data type 217

imagefile assets 186
adding mimetypes for 300
coding templates for 593
example element code 623

implicit objects
and CS 86

importing assets
assets with upload fields 419
attribute editors 435
attributes of type URL 450
basic assets 407
determining how to store multiple field values 418
flex assets 430, 464
flex attributes 436
flex definitions 440
running the BulkLoader utility 475
running XMLPost 422
setting the asset type 413
setting the sites 415, 416
setting workflow 415, 416
source file examples 421
specifying the posting element 413

importing database tables 158

importing ZIP files 159

inheritance
flex assets 306

*iniFile* session variable 167

input table
BulkLoader 465

input types
CS-Direct (basic asset types) 271

InSite Editor
and Flex Assets 742
coding for 737
INSITE.EDIT tag 740
overview 738
template examples 742

Insite Editor
embedded links 740

INTEGER data type 217, 271

Inventory servlet 44, 125

introduction 125
overview 44

invoking
XMLPost 422

IPopulateDataSlice 481

## J

J2EE standard and CS 34

JDBC drivers 464

JDBC-ODBC bridge 464

JSP
and variables 110
ICS Object 82
implicit objects and CS 86

JSP element
adding tag libraries 85

JSP tag libraries
changing default tag libraries 85

## L

link asset
overview 183

linking sessions 794
debugging 800

LIST argument 798

list file
XMLPost 424

list objects
created for assetsets 609

list variables 113
looping through 113

lists
creating 232
looping through lists 97
users in a directory 684

LISTVARNAME argument 798

loading
basic assets 98

LoadSiteTree element 276

LoadTree element 276

log files 176
futuretense.txt 176

logging
dependencies 587

logging out
web site visitors 167

login forms 689

LONGVARCHAR data type 271

LOOP 113

looping through lists 113

Lower range limit field 782, 785

## M

maintaining state 107

management system 35

mapping table 467

MAXCOUNT argument 798

merging property files 162

Microsoft Word
integration with 59

Mimetype field (Stylesheet) 565

MimeType table 300

mimetypes 300
in Burlington Financial 719

Mirror to Server publishing 53, 580

modifyData element 457

modular design
examples 620

moving page assets 570

Mungo tables 203, 204, 606

MungoBlobs table 606

## N

name column
default column for all asset tables 191

Name field
collection assets 557
page assets 567

query assets 562
template assets 507

named associations 187

names
object 98

naming
asset types 269
history attributes 784
visitor attributes 780

ncode column 213

nid column 213

nodes
child 213, 637
IDs 637
parent 213
tree table 213

none dependency 582

NOT NULL constraint 218

nparentid column 213

nrank column 213

Null allowed field 781

NULL constraint 218

NUMERIC data type 218

## O

object ID
as parameter used for loading assets 98, 621

object names 98

object tables 212
creating 220
managing the data in 222
working with 230

oid column 213

otype column 213

ownerid column 204

## P

*p* variable 105

page (online)
defined 497

page assets 66, 183, 565
coding templates for 597
creating 566
defined 497
deleting 572
moving in the site tree 570
placing 569

page caching
CacheManager 584
compositional dependencies 584
guidelines 132
overview 40
SystemItemCache table 215
SystemPageCache table 215

page entries
for template assets 519

page name
defined 497

pagecriteria 514
and the Cache Key 126

pagelets
cacheable 495
defined 497

*pagename* variable 104

pagenames
specifying in an XMLPost configuration file 413

pages
modular design 40
rendering 40

parametric search 57, 206

parent definitions, *see* flex parent definition assets

parents
*See also* flex parent assets
basic assets 187, 296

parser
errors detected 91

*password* variable 105

path
column 194
field 285

Path field

page assets 567

pick asset attribute editor 358

PL status code 560

Place form 570

Place Page workflow function privilege 572

placing page assets 569

posting elements
addData 431
debugging 428
deleteAsset 459
modifyData 457
specifying in an XMLPost configuration file 413

posting, *see* importing assets

PostUpdate element 276, 293

precedence
of variables 112

presentationobject.dtd 352

PreUpdate element 276, 293

previewing
CacheManager 585
CSElement and SiteEntry assets 539
template assets 539

primary key
content tables 214
object tables 212

PRIMARY KEY NOT NULL constraint 218

product discounts 61

product sets 332

promotions
calculating 797
debugging 801
definition 61, 184
duration of 193
recalculating 797

properties
database 227
in XMLPost configuration files 410
loading property files 227
resultsets 260

property
as column and field for basic asset type 264

Property Editor
setting properties 162
starting 161

property files
database properties 227
editing 161
merging 162

property variables 109

proxy servers
XMLPost 410

pubid
defined 561
queries 563

Publication table 68

PublicationTree table 68

publishing
approval 580
approval dependencies 580
defined 495
dynamic vs. static 39
Export to Disk 583
Mirror to Server 580
overview 37, 52

pull-down
attribute editor 358

## Q

queries
code examples 240
Java examples 244, 247, 251
JSP examples 242, 246, 250
overview 256
SystemSQL table 216
tags 232
XML examples 240, 246, 248, 249

query assets 183
coding elements for 595
compositional dependencies 596
defined 558
ExecuteQuery element 559, 596
for collections 561
in Burlington Financial 715, 717
sample element code 631
sample query 559
sharing 563

templates 561

## R

radio buttons
attribute editor 359
input type for basic asset fields 273

range
searchstates 611

Rank field (place pages) 570, 571

ratings
updating before calculating 797

recalculating
segments and promotions 797

recommendations
debugging 801
definition 184
flex assets 790
overview 61
process of developing 790

*referURL* variable 615

registering
a foreign table 225
elements for basic asset types 290
elements with AssetMaker 264

relationships
AssetRelationTree table 187
collections 187
parent-child (basic) 187
parent-child (flex) 197
stored as nodes 213

RemoteContentPost element 408

remotepost, *See* XMLPost

rendering
defined 495
overview 40, 52

*rendermode* variable 105

resargs1 106

resdetails1 106

reserved characters 91

reserved variable names 104

Result of query field (query assets) 563

resultsets
cache flushing 259
load on database 257
overview 42, 256
properties 260

revision tracking
deleting database tables 253
error numbers 178
overview 42, 53

RF status code 560

rich-text search 57, 206, 601

root element
defined 40
for addrow page 235
for page entry 519

Rootelement field (template) 519

rows
updating with CATALOGMANAGER 230

running
BulkLoader utility 475
XMLPost 422

## S

Satellite Server
overview 43

Satellite servlet
caching with 121
overview 44

saving
binary data 780

scalar objects 800

scattering assets 99

scope
and variables 104
searchstates 601

search elements
editing 301

search engines
supported 38

searches
Burlington Financial 715
database 57, 206
directory search 684

full-text 57, 206, 601
parametric 57, 206
rich-text 57, 206, 601

SearchForm element 275

searchstates 206
building 600
overview 57
range constraints 611
resultset caching 259
scope of 601
unfiltered 603
web services 769
wildcards 612

security
overview 42

segments 60
and personalization 778
calculating 797
definition 184
list of 790
overview 60
recalculating 797

*seid* variable 105

select
input type for basic asset fields 272

SelectSummary SQL statement 277

SelectSummarySE SQL statement 277

SELECTTO
example of 258

servlets
BlobServer 36
CatalogManager 36, 230
ContentServer 36
CookieServer 36
DebugServer 36
FlushServer (Satellite Server) 44
HelloCS 37
Inventory (Satellite Server) 44
Satellite (Satellite Server) 44
TreeManager 36

session linking 794
debugging 800

session objects
list of 794

session variables 107, 166
creating 108
login and logout 167
outputting 109
setting 93, 108

sessions 166
example of 167
lifetime of 166
overview 42
tips 172

sharing
assets 555
collection assets 558
CSElement assets 538
query assets 563
SiteEntry assets 538
stylesheet assets 565
template assets 538

shopping carts 800
overview 58

SimpleSearch element 275

site design assets
deleting 556
sharing 555

Site Plan tab 51, 66

site tree 569

*site* variable 105

SiteCatalog table 215
page entries for templates 519

SiteEntry assets 183, 493
calling 589
deleting 538
editing 537
how to invoke one from an element 496
previewing 539
selecting a CSElement for 531
sharing 538
where used 500

SiteGod ACL 692

*sitepfx* variable 105

SitePlanTree table 64, 68, 190
and the Site Plan tab 66
definition 213
example code that displays information from 636

sites
  defined 63, 64
  developing within 66
  example elements 620
  examples of 65
  overview 51
  Publication table 68
  Publication Tree table 68
  setting in an XMLPost configuration file 415, 416
  site plan 66
  SitePlanTree table 64, 68

SMALLINT data type 217, 271

SOAP
  defined 764
  supported version 765

source 560
  adding with asset descriptor file 283
  column 271
  creating for basic assets 299
  field 188

source files, XMLPost
  described 421
  example for attribute editor 435
  flex attribute 439
  flex definition 443
  for flex asset types 433
  identifying them 424
  tags 417

Source table 214

special characters 91

SQL queries
  query assets 559

SQL query field (query assets) 563

SSL authentication 689

start date 561

startdate
  column 193
  field 285

state
  maintaining 107

static publishing
  Export to Disk 53, 583

status 560
  column 192

status codes 560

StatusCode table 214, 560

stylesheet assets 186
  adding mimetypes for 300
  creating 564
  sharing 565

Stylesheet field 565

subtypes
  approval templates 584
  column 193

systable column 216

system design
  page caching guidelines 132

system tables 215, 226

SystemACL table 215

SystemEvent table 215

SystemInfo table 41, 215
  defdir column 219
  systable column 216

SystemItemCache table 215

SystemPageCache table 215

systems
  delivery 35
  development 35
  management 35
  testing 35

SystemSQL table 216

SystemUserAttr table 216

SystemUsers table 216

## T

*tablename* variable 104

tables, *See* database tables

tags
  in XMLPost source files 417
  in XMLPost source files for flex assets 446

template assets 183, 493, 498, 798
  approval templates for Export to Disk 583
  creating 501

default approval templates 537
deleting 538
displaying fields with embedded links 740
editing 537
entry in SiteCatalog table 519
previewing 539
recommendations 799
setting variables for 514, 535
sharing 538
subtypes 584

template column 194

template field
page asset 558, 563, 567

testing system 35

text
input type for basic asset fields 271
text field attribute editor 361

text box, *See* text area

TEXT data type 217

textarea
attribute editor for flex attributes 360
input type for basic asset fields 272

*tid* variable 105

Tile element 276

timeouts
sessions 166, 172

TIMESTAMP data type 271

tree tables 213
creating 222
managing the data in 223
nodes 213
working with 231

tree tabs
flex definitions and hierarchy 200

TreeManager
commands 231
servlet 36

Type field 784

types
constraint 782, 785
data 781, 784
fields (columns) 217
history type 60

## U

unfiltered searchstate 603

UNIQUE NOT NULL constraint 218

UP status code 560

updatedby column 192

updateddate column 192, 560

*updatetype* variable 293, 295

updating
rows with CATALOGMANAGER 230

upload fields 272
and XMLPost 419
examples for basic asset types 284
for basic asset fields 272
stylesheet 565

uploading
asset descriptor files 288

Upper range limit field 782, 785

URL column 219

urlexternaldoc column 195

urlexternaldocxml column 195

URLs 95
adding variables to 106
BlobServer 606
Content Server URL 495
for hyperlinks 613

user management
on delivery system 682

user name 167
default 166
for XMLPost 414
session variable 166

*username* variable 104

users
account creation form 691
USER tags 689

## V

validating
data from input forms 796

VALUE argument 798

VARCHAR data type 271

variables
  and JSP 110
  and scope 104
  appending to URLs 106
  assigning one to another 109
  *authpassword* 105
  *authusername* 105
  best practices 112
  *c* 105
  Cache Criteria (pagecriteria) 514, 535
  *cid* 105
  *context* 105
  counters 114
  creating with Content Server Explorer 106
  *ct* 105, 629
  *currentACL* 105
  displaying variable values 109
  *eid* 105
  *errdetails* 177
  *errno* 105, 177, 178, 616
  evaluating 93, 109
  *ft_ss* 105
  *ftcmd* 104
  HTML forms 107
  in HTML 110
  lists 113
  outputting values 109
  *p* 105
  *pagename* 104
  *password* 105
  precedence of 112
  *referURL* 615
  *rendermode* 105
  reserved names 104
  resolving 94, 111
  retrieving variable values 109
  *seid* 105
  session 107
  session (see session variables)
  setting 93
  *site* 105
  *sitepfx* 105
  *tablename* 104
  *tid* 105
  *updatetype* 293, 295
  *username* 104
  using in an HTML tag 624, 627, 638

visitor attributes 60, 184
  binary 797, 800
  creating 780
  example code 796
  retrieving 801

visitor context, defined 794

visitor data
  collecting 795
  tables 795

visitor data assets 778
  descriptions 780
  development process 779
  testing 787

visitor data collection
  debugging 801

visitor data manager 794
  tags 794

visitor segments 60

visitors
  authentication 688, 695, 720
  collecting data for 795
  elements that collect data for 796
  identifying 794
  IDs 794

VMVISITORALIAS table 794

VMVISITORSCALARBLOB table 795

VMVISITORSCALARVALUE table 795

VMz tables 795

VO status code 560, 617

## W

web services
  complex data types 770
  creating custom web services 769
  defined 764
  generating client code 769
  ILists 769
  predefined functions 768
  process flow 768
  remote procedure calls 770
  required technologies 765
  searchstates 769
  supported data types 768
  using predefined web services 768
  writing a page 770
  writing an element 771
  writing function calls 769

workflow
and page assets 572
assigning to imported assets 415, 416
coding workflow elements 747
overview 53

writing
history definition data 795

WSDL
defined 765
W3C site 765

WSDL files
Content Server 768
creating 773
example 773
generating client code 769
location 768
sections 773

## X

XML
and Satellite Server 122
encoding 90
entities 91
errors 91
special characters 91

XMLPost utility
and flex attributes 436
and upload fields 419
attributes of type URL 450
configuration file example 419
configuration file examples 435, 442, 444, 449
configuration files 410
configuration properties 410
deleting assets 459
determining how to store multiple field values 418
editing flex assets 457
example source files 435, 439
file encoding 421
flex asset types 430
flex definitions 440
proxy servers 410
running it 422
setting the posting element 413
setting the sites 415, 416
setting workflow 415, 416
source file example 421
source file examples 443
source file properties 415
specifying which asset type to import 413
tags for source files 417
troubleshooting 428
under program control 425
username 414

## Y

ZIP files
exporting 157
importing 159