Content Server

Version: 6.3

Developer's Guide

Document Revision Date: Jun. 15, 2011

FatWire
SOFTWARE

FATWIRE CORPORATION PROVIDES THIS PUBLICATION “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 loss of profits, loss of business, loss of use of data, interruption of business, or for indirect, special, incidental, or consequential damages of any kind, 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 © 2006–2011 FatWire Corporation. All rights reserved.

This product may be covered under one or more of the following U.S. patents: 4477698, 4540855, 4720853, 4742538, 4742539, 4782510, 4797911, 4894857, 5070525, RE36416, 5309505, 5511112, 5581602, 5594791, 5675637, 5708780, 5715314, 5724424, 5812776, 5828731, 5909492, 5924090, 5963635, 6012071, 6049785, 6055522, 6118763, 6195649, 6199051, 6205437, 6212634, 6279112 and 6314089. Additional patents pending.

FatWire, Content Server, Content Server Bridge Enterprise, Content Server Bridge XML, Content Server COM Interfaces, Content Server Desktop, Content Server Direct, Content Server Direct Advantage, Content Server DocLink, Content Server Engage, Content Server InSite Editor, Content Server Satellite, and Transact are trademarks or registered trademarks of FatWire, Inc. in the United States and other countries.

iPlanet, *Java*, *J2EE*, *Solaris*, *Sun*, and other Sun products referenced herein are trademarks or registered trademarks of Sun Microsystems, Inc. *AIX*, *IBM, WebSphere,* and other IBM products referenced herein are trademarks or registered trademarks of IBM Corporation. *WebLogic* is a registered trademark of BEA Systems, Inc. *Microsoft*, *Windows* and other Microsoft products referenced herein are trademarks or registered trademarks of Microsoft Corporation. *UNIX* is a registered trademark of The Open Group. Any other trademarks and product names used herein may be the trademarks of their respective owners.

This product includes software developed by the Apache Software Foundation (http://www.apache.org/) and software developed by Sun Microsystems, Inc. This product contains encryption technology from Phaos Technology Corporation.

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) Cuba, Libya, North Korea, Iran, Iraq, Sudan, Syria, or 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.”

Content Server Developer’s Guide
Document Revision Date: Jun. 15, 2011
Product Version: 6.3

FatWire Technical Support
www.fatwire.com/Support

FatWire Headquarters
FatWire Corporation
330 Old Country Road
Suite 303
Mineola, NY 11501
www.fatwire.com

Table of

Contents

About This Guide 21
Who Should Use This Guide 21
How This Guide Is Organized 21
Related Publications 22

Part 1. Overview

1 Overview of Content Server 25
Content Server Product Family 26
Product Summary 26
Third-Party Components 29
J2EE Compliance 29
Content Server Systems 29
The Content Server Core 31
Servlets and Java APIs 31
Page-Generation Components 34
Database Management Functions 36
Sessions and Cookies 37
Event Management Features 38
Satellite Server 38
Handling the HTTP Requests 38
Satellite Server Servlets and APIs 38
CS-Direct 40
Basic Asset Model 40
Standard CS Interface 43
Content Server Clients (Interface Options) 44
Sites and the Site Plan 44
Sample Sites 44

Template, CSElement, and SiteEntry Assets 45
Custom CS-Direct XML and JSP Tags. 45
Approval and Publishing. 45
Revision Tracking 46
Workflow 46
Searching and Search Engines 47
CS-Direct Advantage. 47
Flex Asset Model 47
Assetsets and Searchstates: Searching the Online Site 50
Shopping Carts and Commerce Context 50
GE Lighting Sample Site 51
Custom CS-Direct Advantage XML and JSP tags 51
Content Server Clients (Interface Options) 52
Engage 53
Visitor Data and Segments 53
Recommendations. 53
Promotions 54
Persistent, Linked Visitor Sessions. 54
Custom Engage XML and JSP tags 54
Content Server Portal Interface 55

2 Overview of Sites 57
Content Management Sites 58
Online Sites 59
Developers and the Content Management Site 60
Sites and the Site Plan 60
Sites and the Database 62

3 Content Server Development Process. 63
Step 1: Set Up the Team 64
Step 2: Create Functional and Design Specifications 65
Functional Requirements 65
Page Design. 65
Caching Strategy 65
Security Strategy (Access Control). 65
Separate Format from Content (Elements from Assets) 66
Data Design. 67
Step 3: Set Management System Requirements 68
Step 4: Implement the Data Design 69
Step 5: Build the Online Site 69
Step 6: Set Up the Management System 70
Import Content as Assets 70
Import Catalog Data and Flex Asset Data 70
Instruct the Editorial Team About Site Design. 70
Step 7: Set Up the Delivery System 71

Step 8: Publish to the Delivery System 71

Part 2. Programming Basics

4 Programming with Content Server 75
Choosing a Coding Language 76
The Content Server Context 76
The ICS Object 76
The FTCS tag 77
Content Server JSP 77
Content Server Standard Beginning 77
JSP Implicit Objects 80
Syntax 80
Actions 80
Declarations 81
Scriptlets and Expressions 81
JSP Directives 81
Content Server Tag Libraries 82
Content Server XML 84
Content Server Standard Beginning 84
XML Entities and Reserved Characters 85
XML Parsing Errors 85
Content Server Tags 86
Tags That Create the Content Server Context 87
Tags That Handle Variables 87
Tags That Call Pages and Elements 88
Tags That Create URLs. 89
Tags That Control Caching 90
Tags That Set Cookies 90
Programming Construct Tags 91
Tags That Manage Compositional and Approval Dependencies 91
Tags That Retrieve Information About Basic Assets 92
Tags That Create Assetsets (Flex Assets) 94
Tags That Create Searchstates (Flex Assets) 96
Variables 98
Reserved Variables 98
PageCriteria Variables 100
Setting Regular Variables 100
Setting Session Variables 102
Working With Variables 103
Variables and Precedence 106
Best Practices with Variables 106
Other Content Server Storage Constructs 107

Built-ins . . . 107
Lists . . . 107
Counters . . . 108
Values for Special Characters . . . 109

5 Page Design and Caching . . . 111
Modular Page Design . . . 112
Caching . . . 113
Content Server Caching . . . 113
BlobServer and Caching . . . 113
Satellite Server Caching . . . 114
Viewing the Contents of the Satellite Server Cache . . . 119
CacheManager . . . 120
The SiteCatalog Table . . . 120
The Cache Key . . . 120
Caching Properties . . . 121
Double-Buffered Caching . . . 122
Implementing Double-Buffered Caching . . . 125
Setting cscacheinfo . . . 127
Coding for Caching . . . 127
Caching and Security . . . 128

6 Advanced Page Caching Techniques . . . 131
Overview . . . 132
Configuring the Content Server Cache . . . 133
Configuring the Blob Server Cache . . . 135
Configuring the Satellite Server Cache . . . 136
CacheInfo String Syntax . . . 137
CacheInfo String: First Part . . . 137
CacheInfo String: Second Part . . . 137
Page Timeout . . . 138
Absolute Moment in Time . . . 138
TimePattern . . . 138
Wildcard . . . 138
Blank . . . 138

7 Content Server Tools and Utilities . . . 139
Content Server Explorer . . . 140
Connecting to a Content Server Database . . . 140
CatalogMover . . . 141
Starting CatalogMover . . . 141
Connecting to Content Server . . . 142
CatalogMover Menu Commands . . . 143
Exporting Tables . . . 144
Importing Tables . . . 146

Command Line Interface 148
Property Editor 150
Starting the Property Editor 150
Setting Properties 150
Merging Property Files 151
Page Debugger 152
XMLPost 152

8 Sessions and Cookies 153
What Is a Session? 154
Session Lifetime 154
Session Variables Maintained by Content Server 154
Logging In and Logging Out 155
Sessions Example 155
FeelingsForm Element 155
SetFeeling Element 156
Meat Element 156
What Is a Cookie? 157
CookieServer 157
Cookie Tags 157
Cookie Example 158
Start.xml 158
ColorForm 159
CreateCookie 159
DisplayWelcome 159
Running the Cookie Example 160
Tips and Tricks 160
Satellite Server Session Tracking 160
Flushing Session Information 160

9 Error Logging and Debugging 163
Overview 164
Error Log File Contents 165
Additional Error Message Locations 168
XML Syntax and Runtime Error Checking 169
Debugging Properties 170
Using Error Codes with Tags 171
Tag Examples Using Error Codes 172
Error Number Rules 173
Using the Page Debugger 173
Invoking the Page Debugger 174
Page Debugger Commands 176
Debugging Content Server Applications 179
Debugging Engage 179
Property Messages 181

Part 3. Data Design

10 Data Design: The Asset Models 185
Asset Types and Asset Models 186
Two Data Models 186
Default (Core) Asset Types. 186
Which Asset Model Should You Use to Represent Your Content? 189
The Basic Asset Model 190
Basic Asset Types from the Burlington Financial Sample Site 190
Relationships Between Basic Assets. 191
Category, Source, and Subtype 192
Basic Asset Types and the Database. 194
The Flex Asset Model 199
The Flex Family 199
Sample Site Flex Families. 200
Flex Attributes. 202
Flex Parents and Flex Parent Definitions 203
Flex Assets and Flex Definition Assets 205
Flex Filters 206
Flex Families and the Database 208
Assetsets and Searchstates. 211
Search Engines and the Two Asset Models. 212
Tags and the Two Asset Models 213
Summary: Basic and Flex Asset Models 214
Where the Asset Models Intersect 214
Where the Asset Models Differ 214
Summary: Asset Types 215

11 The Content Server Database 217
Types of Database Tables 218
Object Tables 218
Tree Tables 219
Content Tables 220
Foreign Tables. 220
System Tables 221
Identifying a Table’s Type 222
Types of Columns (Fields) 223
Generic Field Types 223
Database-Specific Field Types 224
Indirect Data Storage with the Content Server URL Field 225
Creating Database Tables 226
Creating Object Tables 226
Creating Tree Tables. 228
Creating Content Tables 229
Registering a Foreign Table 231

How Information Is Added to the System Tables 232
Property Files and Remote Databases 233
Property Files for Remote Databases 234
Accessing the Property File for a Remote Database 234

12 Managing Data in Non-Asset Tables 235
Methods and Tags 236
Writing and Retrieving Data 236
Querying for Data 238
Lists and Listing Data 238
Coding Data Entry Forms 240
Adding a Row 240
Deleting a Row 243
Querying a Table 246
Querying a Table with an Embedded SQL Statement 254
Managing the Data Manually 258
Deleting Non-Asset Tables 259

13 Resultset Caching and Queries 261
Overview 262
Database Queries 262
How Resultset Caching Works 262
Reducing the Load on the Database 263
How Content Server Identifies a Resultset 263
Specifying the Table Name 264
SELECTTO 264
EXECSQL 264
CALLSQL 264
Search Forms in the Content Server Interface 265
Query Asset 265
SEARCHSTATE 265
Flushing the Resultset Cache 265
Enabling Resultset Caching 266
Table-Specific Properties 267
Planning Your Resultset Caching Strategy 267
Summary 267

14 Designing Basic Asset Types 269
The AssetMaker Utility 270
How AssetMaker Works 270
Asset Descriptor Files 274
Columns in the Asset Type's Database Table 276
Elements and SQL Statements for the Asset Type 280
Creating Basic Asset Types 284
Overview 284

Before You Begin 285
Step 1: Code the Asset Descriptor File 286
Step 3: Upload the Asset Descriptor File in to Content Server 294
Step 4: Create the Asset Table (continued from Step 3) 295
Step 5: Configure the Asset Type 296
Step 6: Enable the Asset Type on Your Site 297
Step 7: Fine-Tune the Asset Descriptor File 298
Step 8: (Optional) Customize the Asset Type Elements 298
Step 9: (Optional) Configure Subtypes 300
Step 10: (Optional) Configure Association Fields 301
Step 11: (Optional) Configure Categories 303
Step 12: (Optional) Configure Sources 304
Step 13: (Conditional) Add Mimetypes 305
Step 14: (Optional) Edit Search Elements to Enable Indexed Search 306
Step 15: Code Templates for the Asset Type 306
Step 16: Move the Asset Types to Other Systems 307
Deleting Basic Asset Types 308
Images and eWebEditPro 308

15 Designing Flex Asset Types 309
Design Tips for Flex Families 310
Visitors on the Delivery System 310
Users on the Management System 310
How Many Attribute Types Should You Create? 311
Designing Flex Attributes 311
How Many Definition Types Should You Create? 313
Designing Parent Definition and Flex Definition Assets 313
Summary 315
The Flex Family Maker Utility 315
The Flex Asset Elements 315
Creating a Flex Asset Family 316
Overview 316
Before You Begin 317
Step 1: Create a Flex Family or a New Flex Family Member 317
Step 2: Enable the New Flex Asset Types 319
Step 3: Create Flex Attributes 321
Step 4: (optional) Create Flex Filter Assets 325
Step 5: Create Parent Definition Assets 328
Step 5: Create Flex Definition Assets 330
Step 6: Create Flex Parent Assets 332
Step 8: Code Templates for the Flex Assets 334
Step 9: Test Your Design (Create Test Flex Assets) 334
Step 10 (optional): Create Flex Asset Associations 334
Step 11: Move the Asset Types to Other Systems 335
Editing Flex Attributes, Parents, and Definitions 335

Editing Attributes 335
Editing Parent Definitions and Flex Definitions 336
Editing Parents and Flex Assets 336
Using Product Sets 337
What Is a Product Set? 337
Creating Product Sets 337
Custom Filter Classes or Transformation Engines 338
Registering a New Filter Class 338
Registering a New Transformation Engine 338

16 Designing Attribute Editors 341
Overview 342
The presentationobject.dtd File 343
The Attribute Editor Asset 345
The Attribute Editor Elements 351
Creating Attribute Editors 355
Customizing Attribute Editors 357
Example: Customized Attribute Editor 357
Editing Attribute Editors 362

17 Importing Assets of Any Type 363
The XMLPost Utility 364
Overview 364
XMLPost Configuration Files 366
Configuration Properties for XMLPost 366
Configuration Properties for the Posting Element 369
Configuration Properties for the Source Files 371
Sample XMLPost Configuration File 375
XMLPost Source Files 377
Sample XMLPost Source File 377
XMLPost and File Encoding 377
Using the XMLPost Utility 378
Before You Begin 378
Running XMLPost from the Command Line 378
Running XMLPost as a Batch Process 380
Running XMLPost Programmatically 380
Customizing RemoteContentPost and PreUpdate 381
Setting a Field Value Programmatically 381
Setting an Asset Association 382
Troubleshooting XMLPost 383
Debugging the Posting Element 383

18 Importing Flex Assets 385
Overview 386
Importing the Data Structure Flex Asset Types 386

Importing the Flex Assets 386
Importing Flex Assets: The Process 386
XMLPost and the Flex Asset Model 388
Internal Names vs. External Names 389
Importing the Structural Asset Types in the Flex Model 389
Attribute Editors 389
Flex Attributes. 391
Flex Definitions and Flex Parent Definitions 395
Flex Parents. 399
Importing Flex Assets with XMLPost. 400
Configuration File Properties and Source File Tags for Flex Assets 401
Sample Flex Asset Configuration File for addData 404
Sample Flex Asset Source File for addData 406
Sample Flex Asset Configuration File for RemoteContentPost 409
Sample Flex Asset Source File for RemoteContentPost. 410
Editing Flex Assets with XMLPost 412
Configuration Files for Editing Flex Assets 412
Source Files for Editing Flex Assets. 412
Deleting Assets with XMLPost 414
Configuration Files for Deleting Assets 414
Source Files for Deleting Assets. 415

19 Importing Flex Assets with the BulkLoader Utility 417
Overview of BulkLoader 418
BulkLoader Features. 418
How BulkLoader Works 418
Using the BulkLoader Utility 419
Importing Flex Assets from Flat Tables 420
Step 1: Use XMLPost to Import Structural Assets 421
Step 2: Create the Input Table (Data Source) 421
Step 3: Create the Mapping Table. 423
Step 4: Create the BulkLoader Configuration File 424
Step 5: Run the BulkLoader Utility 431
Step 6: Review Feedback Information 432
Step 7: Approve and Publish the Assets to the Delivery System 432
Importing Flex Assets Using a Custom Extraction Mechanism 432
IDataExtract Interface 433
IPopulateDataSlice 437
IFeedback Interface. 441
Approving Flex Assets with the BulkApprover Utility 442
Creating a Configuration File 442
Using BulkApprover 444

Part 4. Site Development

20 Creating Template, CSElement, and SiteEntry Assets 449
Pages, Pagelets, and Elements 450
Elements, Pagelets, and Caching 450
Calling Pages and Elements 450
Page vs. Pagelet 451
CSElement, Template, and SiteEntry Assets 452
Template Assets 453
CSElement Assets 454
SiteEntry Assets 454
What About Non-Asset Elements? 455
Creating CSElement Assets 455
Before You Begin 455
Step 1: Name and Describe the CSElement Asset 455
Step 2: Configure the Element 457
Step 3: Save and Inspect the CSElement 459
Creating SiteEntry Assets 461
Creating Template Assets 465
Step 1: Name and Describe the Template Asset 465
Step 2: Configure the Page Name 467
Step 3: Configure the Element 468
Step 4: Save and Inspect the Template 469
Managing Template, CSElement, and SiteEntry Assets 471
Designating Default Approval Templates (Export to Disk Only) 471
Editing Template, CSElement, and SiteEntry Assets 472
Sharing Template, CSElement, and SiteEntry Assets 472
Deleting Template, CSElement, and SiteEntry Assets 472
Previewing Template, CSElement, and SiteEntry Assets 473

21 Creating Templates with InSite Templating 475
Overview 476
Designing Slots 478
Choosing Content for Rendered Slots 479
Placing Content 479
Tracking Changes to Master Pages 480
Creating a Master Template 481
Testing Master Templates 482
Pre-requisites 482
Procedures 482

22 Creating Collection, Query, Stylesheet, and Page Assets 487
Previewing Assets 488
Approving Assets 488
Sharing Assets 489

Deleting Assets 489
Collection Assets 490
Before You Begin 490
Creating Collection Assets 491
Sharing Collection Assets 492
Query Assets 492
Query Assets and Other Assets 492
How the Query Is Stored 493
Commonly Used Fields for Queries 493
Before You Begin 495
Creating Query Assets 496
Sharing Query Assets 497
Previewing and Approving Query Assets 497
Stylesheet Assets 497
Creating Stylesheet Assets 498
Sharing Stylesheet Assets 499
Page Assets 499
Creating a Page Asset 500
Placing Page Assets 502
Moving Page Assets in the Site Tree 503
Placing Page Assets and Workflow 504
Editing Page Assets 505
Deleting Page Assets 505

23 Coding Elements for Templates and CSElements 507
About Dependencies 508
The Publishing System and Approval Dependencies 508
Page Generation and Compositional Dependencies 512
About Coding to Log Dependencies 513
ASSET.LOAD and asset:load 513
The ASSETSET (assetset) Tag Family 514
RENDER.GETPAGEURL and render:getpageurl 515
RENDER.LOGDEP (render:logdep) 515
RENDER.FILTER and render:filter 516
RENDER.UNKNOWNDEPS and render:unknowndeps 517
Calling CSElement and SiteEntry Assets 517
Coding Elements that Display Basic Assets 518
Assets That Represent Simple Content 518
Associations 520
ImageFile Assets or Other Blob Assets 521
Basic Assets That Can Have Embedded Links 521
Collections 521
Query Assets 523
Page Assets 525
About Coding Elements that Display Flex Assets 527

Assetsets 527
Searchstates 528
Assetsets, Searchstates, and Flex Attribute Asset Types 529
Scope 529
Coding Templates That Display Flex Assets 530
Example Data Set for the Examples in This Section 530
Examples of Assetsets with One Product (Flex Asset) 531
Special Cases: Flex Attributes of Type Text, Blob, and URL 533
Examples of Assetsets with More Than One Product (Flex Asset) 536
Creating URLs for Hyperlinks 541
RENDER.GETPAGEURL (render:getpageurl) 541
RENDER.SATELLITEBLOB (render:satelliteblob) 542
RENDER.GETBLOBURL (render:getbloburl) 543
Using the referURL Variable 543
Handling Error Conditions 544
Using the Errno Variable 544
Ensuring that Incorrect Pages Are Not Cached 545

24 Template Element Examples for Basic Assets 547
Example 1: Basic Modular Design 548
First Element: Home 549
Second Element: MainStoryList 549
Third Element: LeadSummary 551
Fourth Element: TeaserSummary 551
Back to LeadSummary 552
Back to MainStoryList 552
Back to Home 553
Example 2: Coding Links to the Article Assets in a Collection Asset 553
First element: SectionFront 553
Second element: PlainList 554
Example 3: Using the ct Variable 555
First Element: SectionFront 556
Second Element: TextOnlyLink 557
ColumnistFront 558
Example 4: Coding Templates for Query Assets 559
First Element: Home 559
Second Element: WireFeedBox 560
Third Element: ExecuteQuery 561
Back to WireFeedBox 561
Example 5: Displaying an Article Asset Without a Template 562
First Element: Full 562
Second Element: AltVersionBlock 563
Third Element: EmailFront 563
Example 6: Displaying Site Plan Information 564
First Element: Home 564

Second Element: SiteBanner 564
Third Element: TopSiteBar 565
Back to SiteBanner 567
Example 7: Displaying Non-Asset Information 567
First Element: Home 567
Second Element: ShowMainDate 567

25 User Management on the Delivery System 569
The Directory Services API 570
Entries 570
Hierarchies 570
Groups 570
Directory Services Tags 571
Directory Operations 571
Error Handling 574
Troubleshooting Directory Services Applications 575
Controlling Visitor Access to Your Online Sites 576
ACL Tags 576
User Tags 577
Content Server and Encryption 577
Creating Login Forms 577
Prompt for Login (PromptForLogin.xml) 577
Root Element for the Login Page 578
Creating User Account Creation Forms 579
PromptForNewAccount 579
Root Element for the CreateAccount Page 580
Visitor Access in the Burlington Financial Sample Site 583
Membership Table 583
Users and Passwords 583
Member Accounts 583
Membership Processing Elements 583

26 The HelloAssetWorld Sample Site 585
Overview 586
HelloAssetWorld Templates 586
HelloAssetWorld Asset Types 587
Modified Asset Types 587
The HelloArticle Asset Type 587
The HelloImage Asset Type 588
HelloAssetWorld Templates 590
The HelloArticle Template 590
The HelloCollection Template 593
The HelloPage Template 596
The HelloQuery Asset 598

27 The Burlington Financial Sample Site 599
Overview 600
Navigation Features 601
Breadcrumbs 602
Best Practices 603
Searching 603
Keywords 603
Hot Topics 603
Topic Directory 604
Related Stories 605
Text-Only Versions 606
Plain Text Parallel Site 606
E-mail This Story 607
AssetMaker Asset Types 607
Mimetype 607
Collections of Collections 608
Membership 608
Wire Feed 609
Featured Funds 609
Fund Finder 609
Page Cache Parameters 610

Part 5. Management System Features

28 Customizing the User Interface 613
Overview of the Tree 614
Loading the Tree Tabs 614
Refreshing the Tree 621
Trees and Security 622
Tree Error Logging 622

29 Coding for the InSite Editor 623
Overview 624
The INSITE.EDIT Tag 626
Parameters 626
Syntax 627
Supported Data Types and Input Types 628
Template Element Examples 628
Example for Basic Asset 629
Example for Flex Assets 630
Example for an Attribute of Type Blob 631

30 Customizing Workflow 633
Workflow Step Conditions 634
Workflow Actions 635
Step Action Elements 636
Timed Action Elements 638
Deadlock Action Elements 640
Group Deadlock Action Elements 643
Delegation Action Elements 645

Part 6. Web Services

31 Overview of Web Services 649
What Are Web Services? 650
SOAP and Web Services 650
Supported SOAP Version 650
Supported WSDL Version 650
Related Programming Technologies 651

32 Creating and Consuming Web Services 653
Using Predefined Web Services 654
Accessible Information 654
WSDL File Location 654
Process Flow 655
Consider Your Data 655
Generating the Client Interface 655
Writing Client Calls 655
Creating Custom Web Services 656
Process Flow 656
Consider Your Data 656
Creating a Content Server Page 657
Writing a Content Server Element 658
Creating a WSDL File 659
Consuming Web Services 661
Locating the Web Service 661
Gathering Information from the Remote WSDL File 661
Providing Information to Content Server 661

Part 7. Engage

33 Creating Visitor Data Assets 665
About Visitor Data Assets 666

Visitor Attributes 666
History Attributes and History Definitions 666
Segments 666
Categories 667
Developing Visitor Data Assets: Process Overview 667
Creating Visitor Data Assets 668
Creating Visitor Attributes 668
Creating History Attributes 671
Creating History Definitions 674
Verifying Your Visitor Data Assets 675
Approving Visitor Data Assets 676

34 Recommendation Assets 677
Overview 678
Development Process 678
Creating a Dynamic List Element 679

35 Coding Engage Pages 681
Commerce Context and Visitor Context 682
Identifying Visitors and Linking Sessions 682
Collecting Visitor Data 683
Coding Site Pages That Collect Visitor Data 684
Templates and Recommendations 686
Creating Templates for Recommendations 687
Shopping Carts and Engage 688
Debugging Site Pages 688
Session Links 688
Visitor Data Collection 689
Recommendations and Promotions 689

Appendixes

A. Creating a Hierarchical Flex Family 693
Overview 694
Hierarchical Organization 694
Flex Family Specifications 695
Procedures 696
Step 1: Create a Flex Family 696
Step 2: Enable the New Flex Asset Types 697
Step 3: Set Up Content Server's Tree 698
Step 4: Create Flex Attributes 699
Step 5: Create Parent Definition Assets 700
Step 6: Create Flex Parent Assets 702

Step 7: Create Flex Definition Assets 704
Step 8: Create Flex Assets 707
Next Steps 710

B. Content Server URL Assemblers 711
Overview of Content Server URL Assemblers 712
URL Assembly 712
Assembler Discovery and Disassembly 712
Assemblers Installed with Content Server 713
Working with Assemblers 713
Creating Assemblers 713
Registering and Ranking Assemblers 714
Modifying Link Tags 715

C. White Space and Compression 717
White Space and JSP 718
White Space and XML 718
Compression 718
JSP Design 719

Index 721

About This Guide

This guide describes the 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 especially 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, JavaScript 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.

The final part, “Appendixes,” contains supplementary information. One appendix provides a tutorial for creating a hierarchically organized flex family. Other appendixes provide information about working with URL assemblers and implementing whitespace compression techniques.

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* amplifies on functions related to the management of CM sites, users, workflow processes, CS clients, and security.

Related Publications

The FatWire library includes publications written for Content Server users, administrators, and developers. The publications are provided as product manuals with your Content Server installation. They are also posted on the Web at the following url:

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

Check the site regularly for updates.

Other publications, such as case studies and white papers, provide information about Content Server’s feature set and its business applications. To obtain these publications, contact `sales@fatwire.com`.

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

Chapter 1

Overview of Content Server

The 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
- 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, which prior to version 6.1 were offered as products (the modules can be optionally installed), 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).
  - Page Debugger, for stepping through the execution of XML and JSP code.

  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 standard Content Server interface, 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 its functionality as well as its 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. Within its own framework the flex asset model also supports flat data structures. (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 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 must be manually 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.

- **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.

- **InSite Editor** supports the editing of content directly on the 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.

  Note

  Unlike other clients, InSite Editor is automatically installed by CS-Direct.

- **eWebEditPro** is a third-party HTML editor from Ektron, Inc. Once configured, eWebEditPro can be used by content providers from within InSite Editor.

  Developers can use eWebEditPro 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. Only one version of eWebEditPro can be used by each Content Management installation.

Note

Information in this guide is based on the assumption that your Content Server systems are running the CS-Direct and CS-Direct Advantage modules.

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.

To enhance Content Server's main capabilities, several content-driven products are available:

- 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, 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.
- Content Server Portal Interface, a product that re-interprets the Content Server standard 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 31.

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.

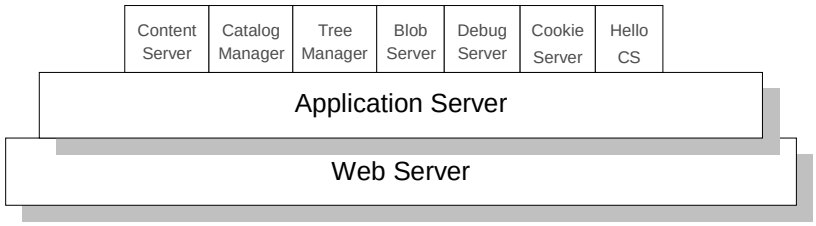
The Content Server Core

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 23, "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."

- 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, Property Editor, and Page Debugger.

The utilities are used together with the Content Server browser-based interface. Chapter 7, "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.

- Information about CS-Explorer usage is given in:
  - Chapter 15, "Designing Flex Asset Types," in the section "Custom Filter Classes or Transformation Engines" on page 338
  - Chapter 20, "Creating Template, CSElement, and SiteEntry Assets"
  - Chapter 23, "Coding Elements for Templates and CSElements"

- Information about CatalogMover usage is given in Chapter 20, “Creating Template, CSElement, and SiteEntry Assets” (pages 453 and 455).
- Information about XMLPost usage is given in various sections of:
  - Chapter 10, “Data Design: The Asset Models”
  - Chapter 14, “Designing Basic Asset Types”
  - Chapter 15, “Designing Flex Asset Types”
  - Chapter 17, “Importing Assets of Any Type”
  - Chapter 18, “Importing Flex Assets” (entire chapter)
  - Chapter 19, “Importing Flex Assets with the BulkLoader Utility”
  - Chapter 22, “Creating Collection, Query, Stylesheet, and Page Assets”
- Information about BulkLoader usage is given in:
  - Various sections of Chapter 18, “Importing Flex Assets”
  - Chapter 19, “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*.
- Information about Page Debugger usage is given in Chapter 9, “Error Logging and Debugging.”

Search Engine Interfaces

The Content Server core does not provide a search engine but it does provide an interface to Verity, a third-party search engine. 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. Because an HTML page is not dynamic, each request can display only one and the same page.	Generated upon request. 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. As a result, it is difficult to modify format and content independently of each other.	Separation of format and content. 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 36, and Chapter 5, "Page Design and Caching." See also "Satellite Server" on page 38.

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 List*, which you can access by going to the URL `e-docs.fatwire.com/CS` and selecting your version of Content Server.

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 11, "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 38.

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 13, "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 25, "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 8, "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 standard interface to the Content Server database, enabling you to easily organize, categorize, manipulate, and maintain your content objects. These content objects are 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
- Standard interfaces for developers (and administrators), as well as content providers
- Client interface options: CS-Desktop, the InSite Editor, and CS-DocLink
- The content management site and the site plan
- The sample sites 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

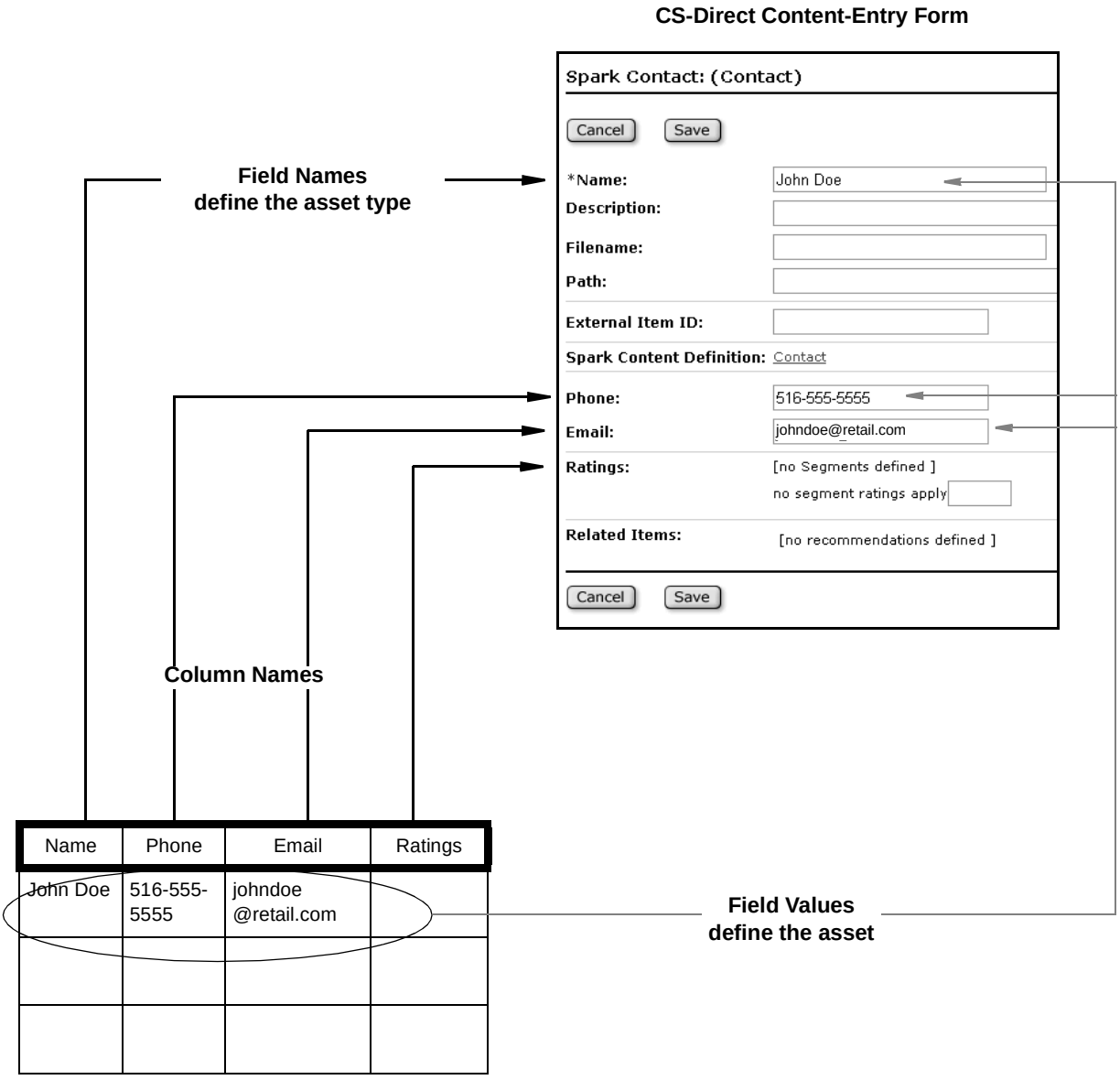
CS-Direct provides a framework for your content, in which that content is stored as objects called **assets** in the Content Server database. The framework is called the **basic asset model**, which allows one primary storage table in the database for each type of asset. When assets are created from 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 an interface to the database tables where the assets are stored. The interface consists 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 is shown in Figure 1, “Asset Types: Content-Entry Forms and Database Tables” on page 41.

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. The field names (specified by developers) define the asset type; the field values (entered by content providers) define the asset.



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, 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.

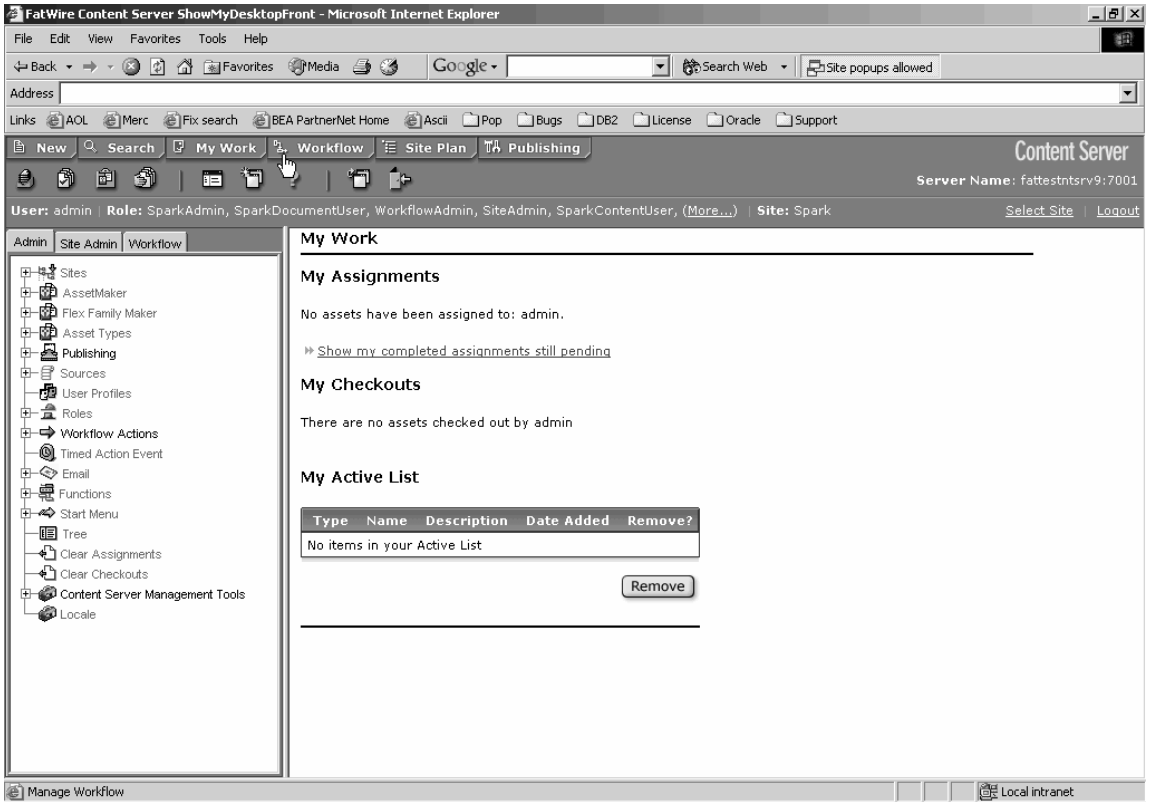
  Like content assets, logic assets use the asset data model, so that you can use CS-Direct content management features (workflow, revision tracking, access control, and so on) to maintain them.

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

- Chapter 10, “Data Design: The Asset Models”
- Chapter 14, “Designing Basic Asset Types”

Standard CS Interface

CS-Direct renders the standard tree-and-workspace interface shown below to support the web-based environment.



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.

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 41. (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.)

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 52.

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 or its 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.

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 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 27, "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 26, "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 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 pagelets and elements 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 20, "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 23, "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 moving 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 publish 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 23, "Coding Elements for Templates and CSElements."

For information about how you approve assets, see the *Content Server User's Guide*.

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 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 default search mechanism is a SQL-based database search. You can replace the default search method with a Verity search engine.

To do so, you install the search engine (typically, as part of the Content Server installation process). When the search engine is present, CS-Direct presents a form on the "Admin" tab that you use to determine which asset types should use the search engine's index-based search, and which (if any) asset types should use the default database search.

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.

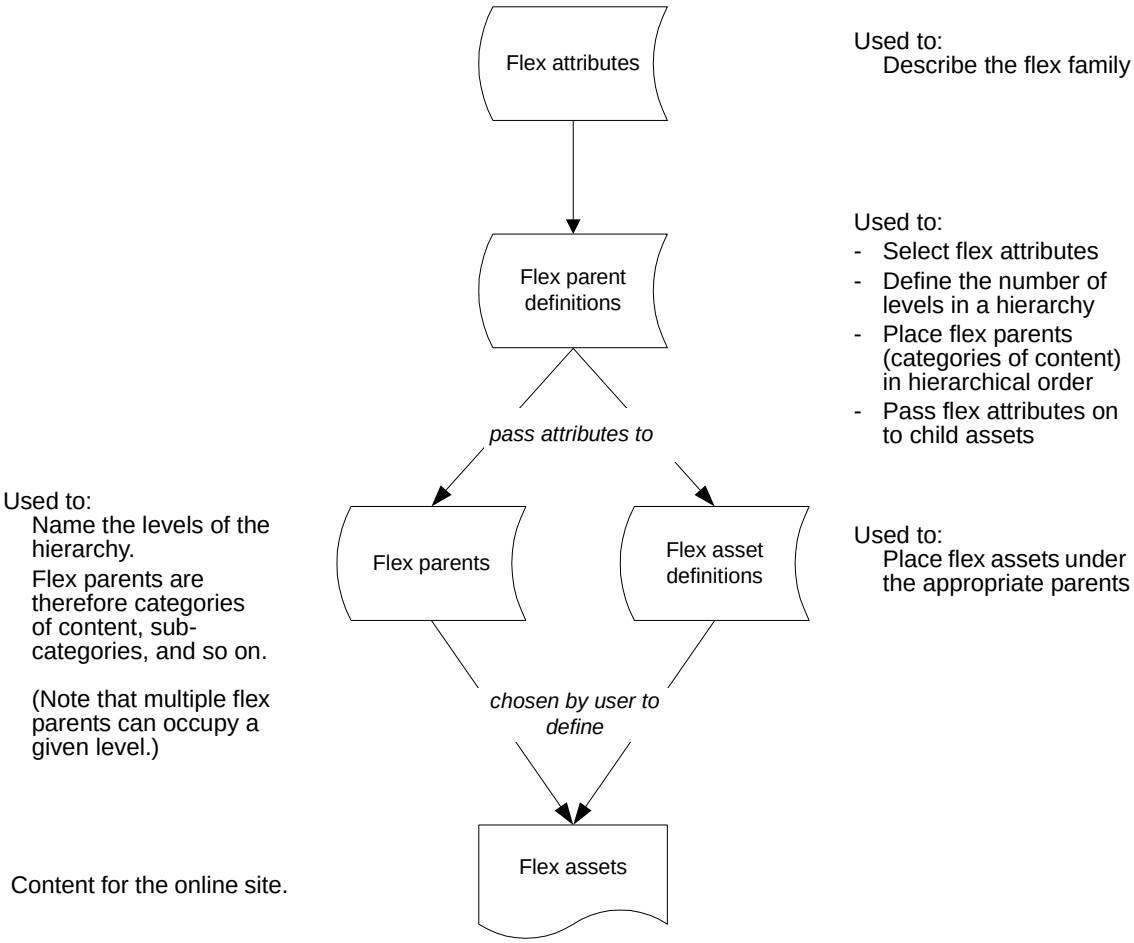
General Information About Flex Families

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 10, "Data Design: The Asset Models"
- Chapter 15, "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 23, "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 10, “Data Design: The Asset Models”
- Chapter 15, “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 23, “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.

- The **InSite Editor** is for content providers who need to make edits to (and perhaps approve) assets in the context of how they actually look when they are rendered in a browser. The InSite Editor allows content providers to work in a continuous preview mode.

  InSite Editor is automatically installed with CS-Direct. Is is also used by CS-Direct Advantage. To enable InSite Editor, you must code your rendering templates to activate InSite Editor for the appropriate asset types.

- **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. Only one version of eWebEditPro can be used by each Content Management installation.

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

For information about coding templates that invoke the InSite Editor, see Chapter 29, "Coding for the InSite Editor."

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, Chapter 33, “Creating Visitor Data Assets.”
- For information about creating segment assets, see the *Content Server 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 34, "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 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 35, "Coding Engage Pages."

Custom Engage XML and JSP tags

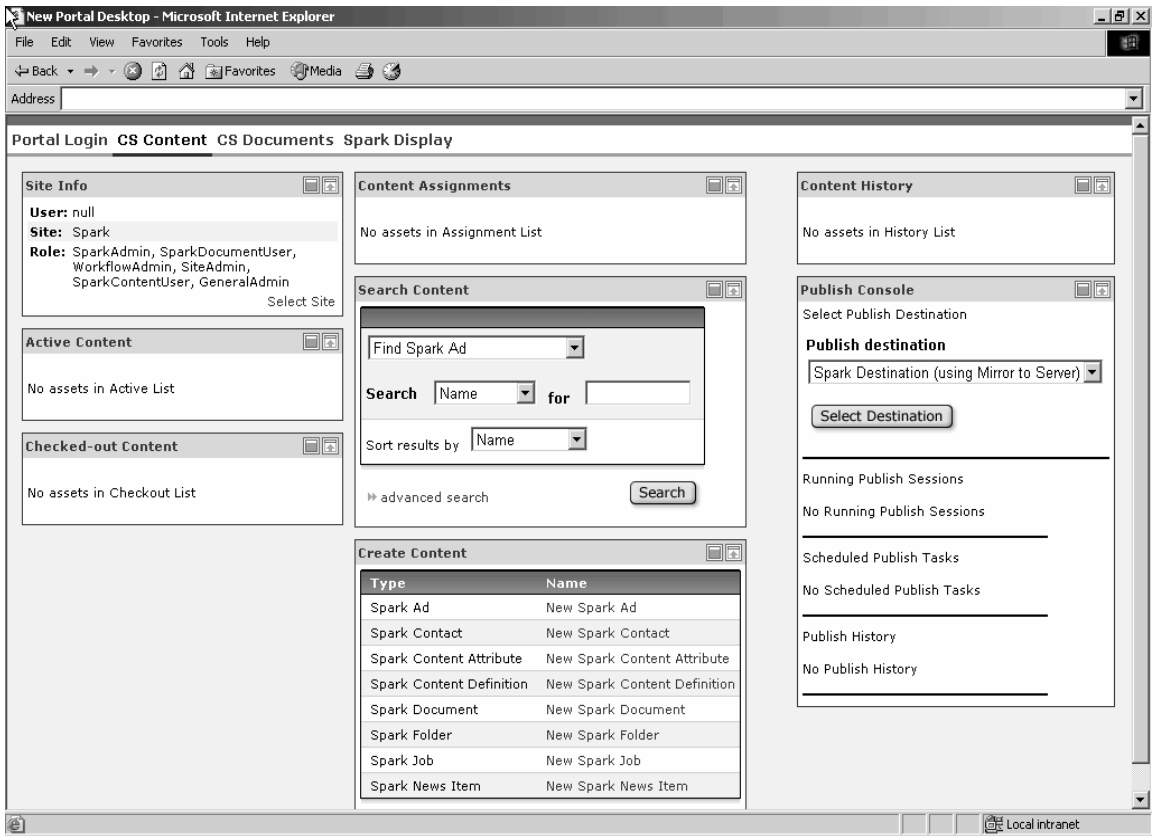
In addition to the tag family that implements the linking of visitor sessions, Engage implements several more custom tag families (both XML and JSP versions.) You use these tags to code pages that collect visitor data and to code the templates for your recommendations.

- For information about coding pages that collect and use visitor data and coding templates for recommendations, see Chapter 35, "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 is a new product, added in the CS 6.1 release, that supports 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

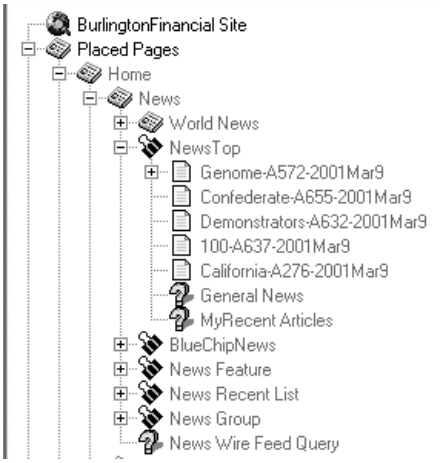
Page assets are site design assets that store references to other assets, organizing your assets according to the design that you and other developers are implementing. During the design phase of your online site, you create page assets, associate other assets with them, and then position the page assets 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 564 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 identify 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, “Advanced Page Caching Techniques”
- Chapter 7, “Content Server Tools and Utilities”
- Chapter 8, “Sessions and Cookies”
- Chapter 9, “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 !=. • The element uses many loops. Loops perform better in JSP than in XML. • 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 82.

For information about commonly used tags that are found in these tag libraries, see "Content Server Tags" on page 86.

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 98.

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 80.

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.
	In addition to the standard directives, you must add two page directives to set the `contentType` and `session` for each Content Server element or template that you create.
	• 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:
	`<%@ page contentType="text/html; charset=UTF-8" %>`
	• Content Server handles sessions for your JSP pages, so you should also disable HTTP sessions for your page. To disable HTTP sessions for the current page, provide the following page directive:
	`<%@ page session="false" %>`

Taglib Directive	Content Server automatically seeds your templates and elements with commonly used taglib directives. 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 section 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> </FTCS>	<ftcs1_0:ftcs> </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 NAME="*variableName*"/>	<ics:getvar name="*variableName*"/>

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

SETVAR (XML)	ics:setvar (JSP)
<SETVAR NAME="*variableName*" VALUE="*variableValue*"/>	<ics:setvar name="*variableName*" 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 NAME="*variableName*" VALUE="*variableValue*"/>	<ics:setssvar name="v*ariableName*" value=*"variableValue*"/>

SETSSVAR sets a session variable.

REPLACEALL (XML)	ics:resolvevariables (JSP)
<REPLACEALL NAME="*variableName*" VALUE="*variableValue*"/>	<ics:resolvevariables name="*variableName*" [output="*variable name*"] [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 98.

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 PAGENAME="*nameOfPageEntry*" [CACHECONTROL="*expiration_date_and_time*"] [ARGS_var1="*value1*"]/>	<render:satellitepage pagename="*nameOfPageEntry*" [cachecontrol="*expiration_date_and_time*"]> <[render:argument name="*variable1*" value="*value1*"]/> </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.

RENDER.SATELLITEPAGE 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 ELEMENTNAME="*nameOfElement*" [ARGS_*var1*="*value*"]/>	<ics:callelement element="*element name*"> <ics:argument name="*argument name*" *value="arg value"*/> </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 OUTSTR="*myURL*" PAGENAME="*SiteCatalogPageEntry*" cid="*IDofAsset*" [p="*IDofParentPage*"] [c="*AssetType*"] [ADDSESSION=“*true*”] [DYNAMIC=“*true*”] [PACKEDARGS=“*stringFromPACKARGStag*”] [ARGS_*xxx*=“*y*”]/>	<render:getpageurl outstr=“*myURL*” pagename="*SiteCatalogPageEntry*" cid="*IDofAsset*" [p="*IDofParentPage*"] [c="*AssetType*"] [addsession=“*true*”] [dynamic=“*true*”] [packedargs=“*stringFromPACKARGStag*”] <[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 SERVICE=“*HTMLtagName*” BLOBTABLE=“*blobTable*” BLOBKEY=“*primaryKeyName*” BLOBWHERE=“*primaryKeyValue*” BLOBCOL=“*columnName*” BLOBHEADERNAMEN=“*headername*” BLOBHEADERVALUEN="mimetype" [ARGS_format1=“*5*”] [CACHECONTROL=“*expirationDateAndTime*”]/>	<render:satelliteblob service=“*HTMLtagName*” blobtable=“*blobTable*” blobkey=“*primaryKeyName*” blobwhere=“*primaryKeyValue*” blobcol=“*columnName*” blobheadernameN=“*headername*” blobheadervalueN="mimetype" [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 Sever 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 545.

Tags That Set Cookies

The following tag sets cookies in Content Server.

satellite.cookie (XML)	satellite:cookie (JSP)
`<satellite.cookie name="cookie_name" value="cookie_value" timeout="timeout" secure="true|false" url="URL" [domain="domain"]/>`	`<satellite:cookie> <satellite:parameter name='name' value='cookie_name'/> <satellite:parameter name='value' value='cookie_value'/> <satellite:parameter name='timeout' value='cookie_timeout'/> <satellite:parameter name='secure' value='true|false'/> <satellite:parameter name='url' value='url'> </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*">
  <THEN>
  tags and/or text
  </THEN>
  <ELSE>
  tags and/or text
  </ELSE>
</IF>
	<ics:if condition="*logical expression*">
	  <ics:then>
	  tags and/or text
	  </ics:then>
	  <ics:else>
	  tags and/or text
	  </ics:else>
	</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*"]
  [COUNT="*LOOP_TIMES*"]
  [LIST="*LIST_NAME*"]
  [UNTIL="*END*"]>
  ...
</LOOP>
	<ics:listloop
	  listname="*some list*"
	  [maxrows="*number of loops*"]
	  [startrow="*start row*"]
	  [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 508.

RENDER.LOGDEP (XML)	render:logdep (JSP)
<RENDER.LOGDEP ASSET="asset name" CID="asset id" C="asset type"/>	<render:logdep asset="asset name" cid="asset id" 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"`
`LISTVARNAME="output list name"`
`LISTDICOL="assetID column"`
`[LISTTYPECOL="assettype column"]`
`[TYPE="asset type"]`
`[ID="asset id"]`
`[VARNAME="output variable"/>`
	`<render:filter list="list name"`
	`listvarname="output list name"`
	`listidcol="assetID column"`
	`[listtypecol="assettype column"]`
	`[type="asset type"]`
	`[id="asset id"]`
	`[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`
`NAME="`*`assetName`*`"`
`TYPE="`*`assetType`*`"`
`OBJECTID="`*`object.id`*`"`
`[FIELD="`*`fieldName`*`"]`
`[VALUE="`*`fieldValue`*`"]`
`[DEPTYPE="EXACT, EXISTS, or GREATER"]/>`
	`<asset:load`
	`name="`*`assetName`*`"`
	`type="`*`assetType`*`"`
	`objectid="`*`object.id`*`"`
	`[field="`*`fieldName`*`"]`
	`[value="`*`fieldValue`*`"`
	`[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 548.

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 NAME="*assetName*" PREFIX=“*variablePrefix*”/>	<asset:scatter name="*assetName*" 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 NAME="*assetName*" FIELD="*fieldName*" [OUTPUT="outputVariable"]/>	<asset:get name="assetName" field="fieldName" [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 NAME=“*assetName*” LIST= “*listName*” [CODE= “*NameOfAssociation*”] [OBJECTTYPE= “*typeOfObject*”] [OBJECTID=“*objectID*”] [ORDER=“*nrank*”]/>	<asset:children name=“*assetName*” list=“*listName*” [code=“*NameOfAssociation*”] [objectype=“*typeOfObject*”] [objectid=“*objectID*”] [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 NAME="*assetsetname*" TYPE="*assettype*" ID="*assetid*" [LOCALE="*localeobject*"] [DEPTYPE="exact|exists|none"] />	<assetset:setasset name="*assetsetname*" type="*assettype*" id="*assetid*" [locale="localeobject"] [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 NAME="assetsetname" [ASSETTYPES="assettype"] [CONSTRAINT="searchstateobject"] [LOCALE="localeobject"] [SITE="siteidentifier"] [DEPTYPE="exact|exists|none"]/>	<assetset:setsearchedassets name="*assetsetname*" [assettypes="*assettype*"] [constraint="*searchstateobject*"] locale="*localeobject*"] [site="*siteidentifier*"] [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 seachstate 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:addsimplestandard constraint 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
- PageCriteria 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.
`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.
`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 555.
`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 Export to Disk, or use 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.

PageCriteria Variables

Content Server's caching mechanism uses variables to help identify different pages in the cache. The term **PageCriteria** designates variables that are used to identify cached pages, which means that the variables are used in the page's cache key.

In order for Content Server to cache and retrieve your web pages properly, you must log any variable that meets the following criteria as a PageCriteria variable:

- The variable affects the rendered page. For example, if your page uses a `color` variable that sets the background color, you must set that variable as PageCriteria.
- You pass the variable from one page to another.

You specify PageCriteria variables in the `resargs` fields of the page's SiteCatalog entry. Variables which meet the preceding criteria but are not specified as PageCriteria will generate an error.

When you save a Template asset, the Content Server Content Applications set the following variables as PageCriteria by default:

- `c`
- `cid`
- `p`
- `rendermode`

For complete information about Content Server caching and PageCriteria, see "PageCriteria and the Cache Key" on page 120.

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 8, “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.Property.verity.path`

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 23, “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 `ft.debug` property in the `futuretense.ini` file to “yes” (`ft.debug=yes`). When this setting is enabled, CS-Direct writes a record of all the variables that are created to the `futuretense.txt` log. For more help with debugging, use the XML Page Debugger utility.

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
- Caching and Security

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:

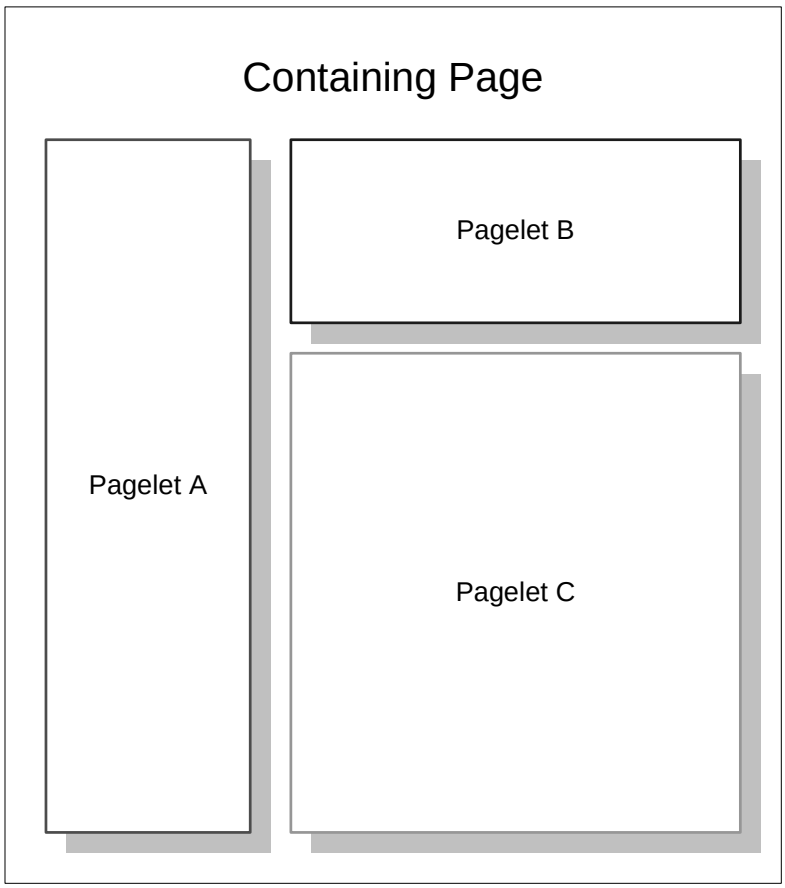


Figure 3: A modular page

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` column of the `SiteCatalog` table. 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 binary large object. 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, rename the `blobtable` parameter in the BlobServer URL to `flushblobtable`, as shown in the following BlobServer URL:

```
http://hostname:port/servlet/BlobServer?blobcol=urlbody
&blobheader=text%2Fcss&blobkey=name&flushblobtable=StyleSheet
&blobwhere=BF-MSIE-Win
```

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 126.

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 137 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 121.
- 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 122.

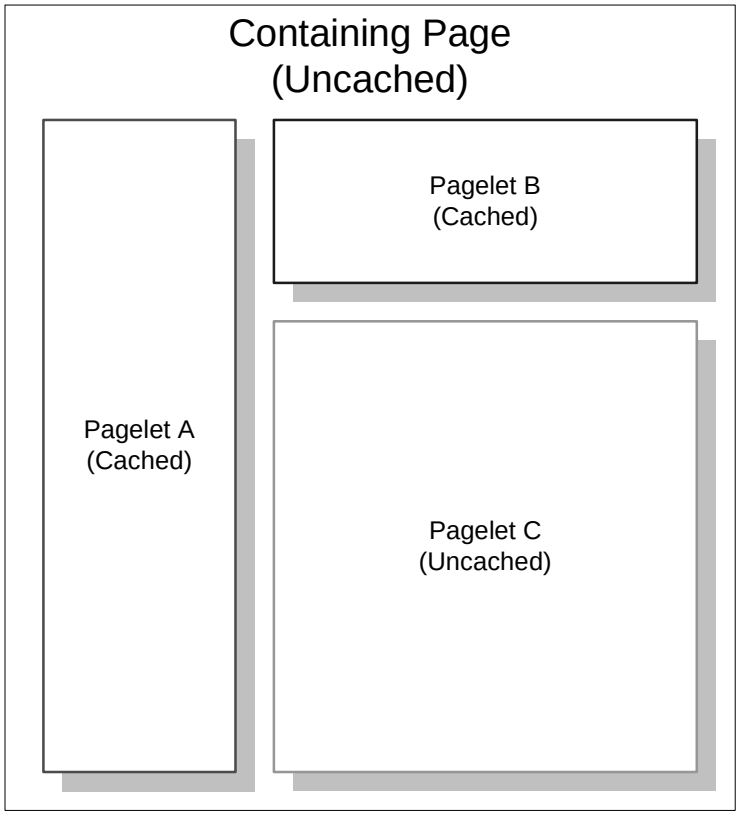
Caching with the Satellite Servlet

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: • `names` - Displays the header information, plus the page names of the pages in the cache. • `keys` - Displays the header information, plus the page names and keys of the items in the cache. 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 465 and "Creating SiteEntry Assets" on page 461.

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` and `resargs` 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 a `PageCriteria` variable in one of the `SiteCatalog` table's `resargs` fields. 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

`PageCriteria=color1,color1&color1=red&color2=blue` in the `resargsl` field.

If you are running CS-Direct, the standard CS-Direct variables are set as `PageCriteria` in the `resargs1` field.

Content Server and Satellite Server use the parameters that are passed 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 generated is the same. For example:

```
http://mysatellite:1234/servlet/
ContentServer?pagename=myPage&color=red
```

calls a different page than:

```
http://mysatellite:1234/servlet/
ContentServer?pagename=myPage&color=blue
```

whether or not the content generated is the same. Values passed in by URL override values set in `PageCriteria`. For example, if you have `myPage's PageCriteria` set to `PageCriteria=red,blue`, and you pass in a value of `green, green` will go into `myPage's` cache key, not `red,blue`.

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.SystemPageCacheSz`, 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.

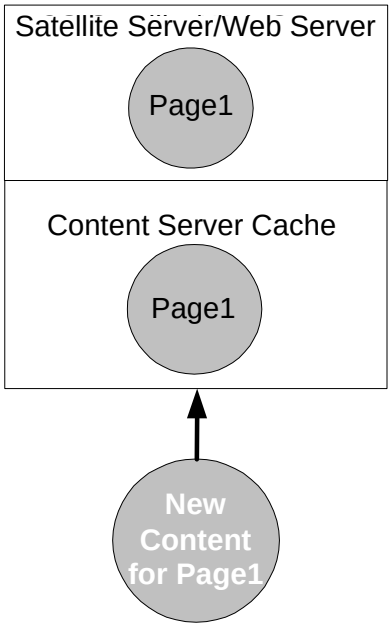
Double-Buffered Caching

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.

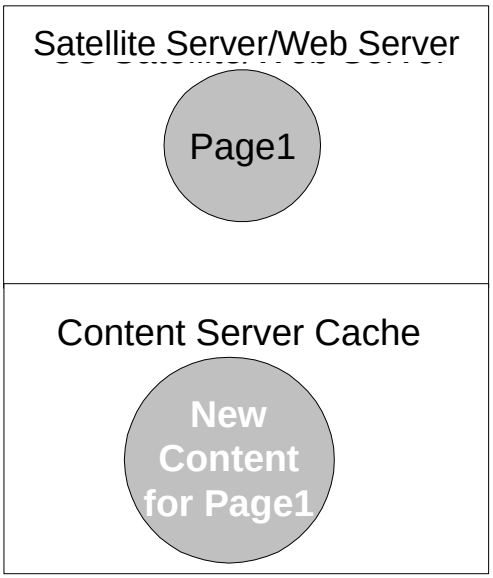
You can implement a similar caching strategy if you are running the Content Server core and Satellite Server without any of the other CS modules or products, by using the CacheManager Java API. For more information about the CacheManager Java API, see the *Content Server Javadoc.*

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:

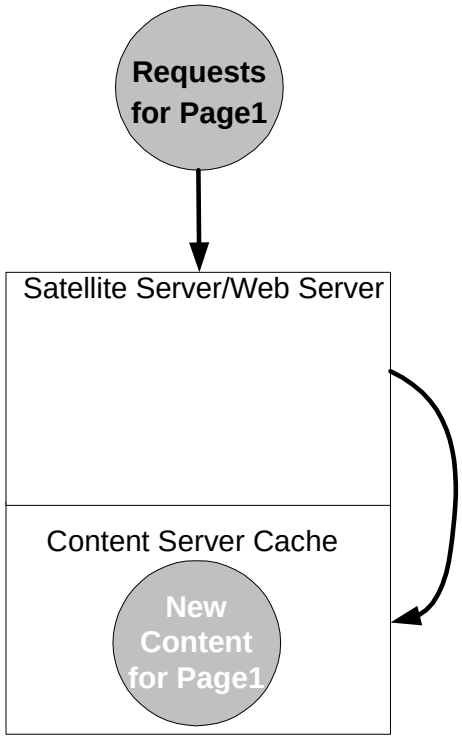


- Content providers publish updated assets to the delivery system. CacheManager checks the cache tracking tables to see which cached items are affected by the updated assets.

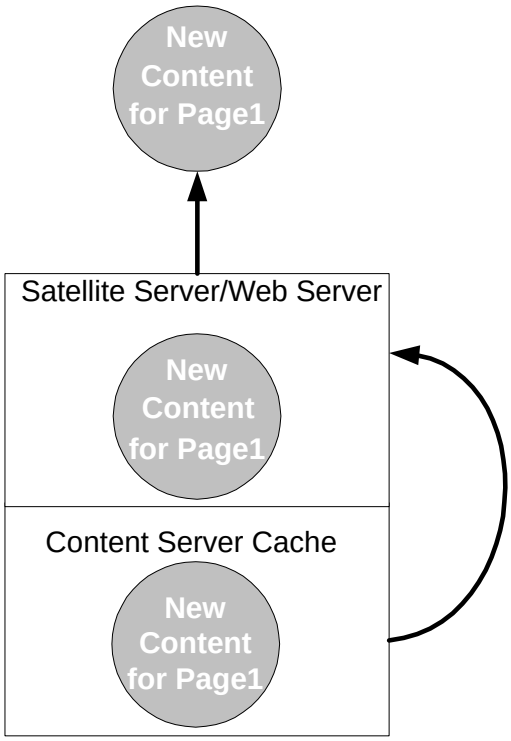


- CacheManager flushes the outdated `Page1` from the Content Server cache, then reloads the Content Server cache with the updated `Page1`.

  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 112. 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 `cachecriteria` field of the `SiteCatalog` table
- Code your elements with Satellite tags

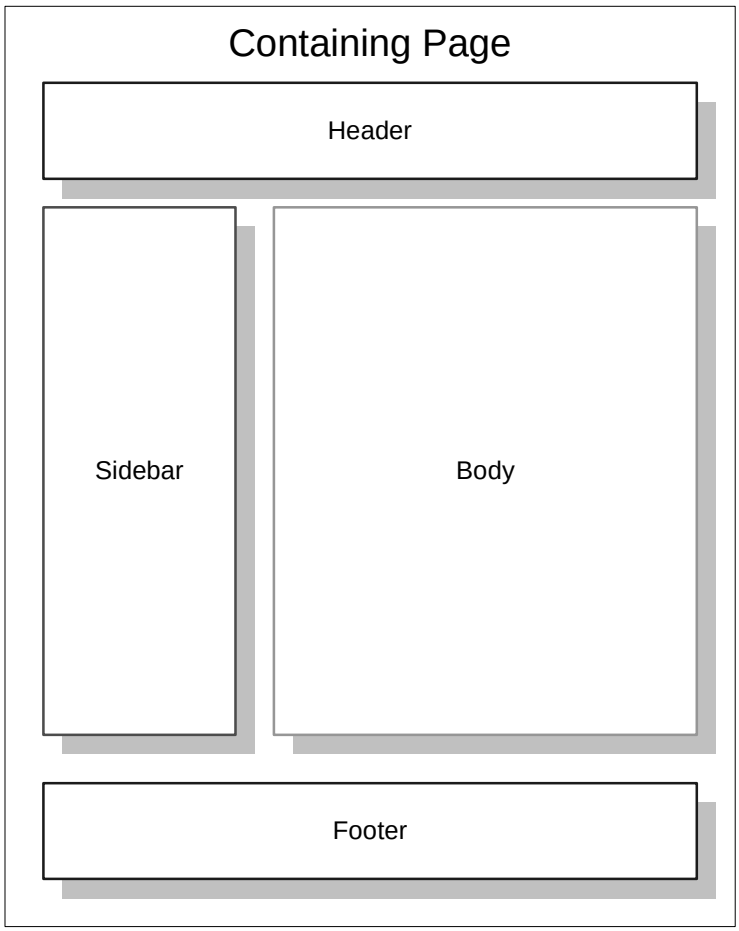
Pagelet Caching Strategies

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. • If the pagelet does not contain logic that requires evaluation to work.	• If the content changes frequently. • If the content must be "real time." • 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 13, "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 465.

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

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:

- CacheInfo String Syntax
- 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 133 (Content Server cache), page 135 (BlobServer cache), and page 136 (SatelliteServer 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 133 (Content Server cache) and page 135 (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 133 (Content Server cache), page 136 (BlobServer cache), and page 136 (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.SystemPageCacheSz` property in the `futuretense.ini` file. Setting `cs.SystemPageCacheSz` to a negative value disables the size restriction. 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). Setting a value of 0 will cause all entries to be added and then promptly removed. However, this should be avoided.

  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 137.

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 122.

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 135.

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 value of 0 indicates that the page will expire immediately. A negative value means that the page should never expire, and 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

Starting with CS 6.1.0, the TimePattern format is now 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.Time Pattern.

This document does not describe valid TimePattern syntaxes in detail. For more information, consult the TimePattern JavaDoc for more information.

In general though, 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 7

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
- Page Debugger
- 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.

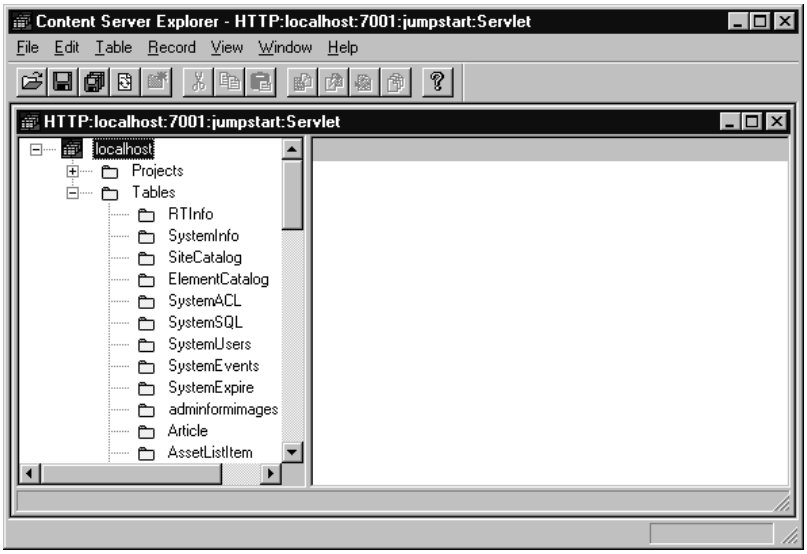
   Password – Your Content Server password. (Depending on your site security, it may not be necessary to enter a name and password.)

   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 148.

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

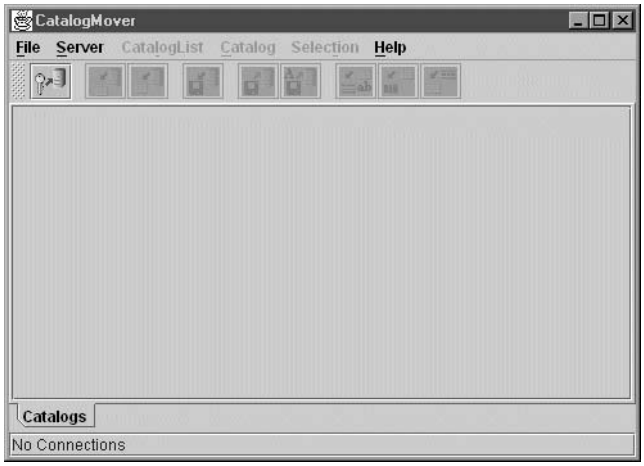
Execute the following scripts at the MS DOS prompt or in a UNIX shell:

- 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. To merge Content Server property (`.ini`) files, enter the names of two property files separated by a semicolon in the **'inifile'(s)** field. If you do not want to merge property files, leave this field blank. (For more information, see "CatalogMover Menu Commands" on page 143.)
6. 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.`
7. Click **Connect**.

CatalogMover Menu Commands

CatalogMover includes the following menu commands:

File Menu

- **Exit** – Disconnect from Content Server and close CatalogMover.

Server Menu

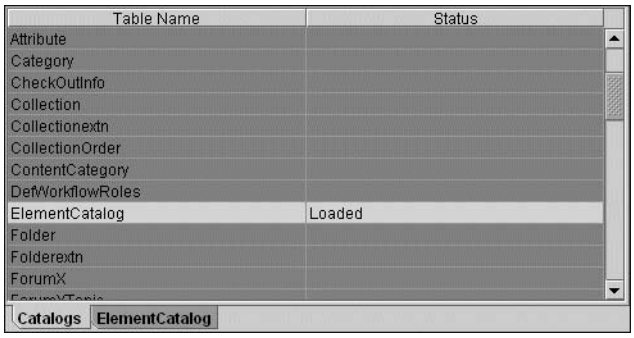
- **Connect** – Display the Connect to Server dialog box.
- **Reconnect** – Display the Connect to Server dialog box and renew the current Content Server connection.
- **Disconnect** – Disconnect from Content Server.
- **Purge Temporary Tables** – Purge imported tables before committing.
- **Commit Individual 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:



  Cick 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 142.
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.

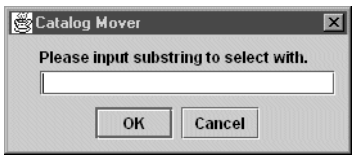
Selecting Rows for Export

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

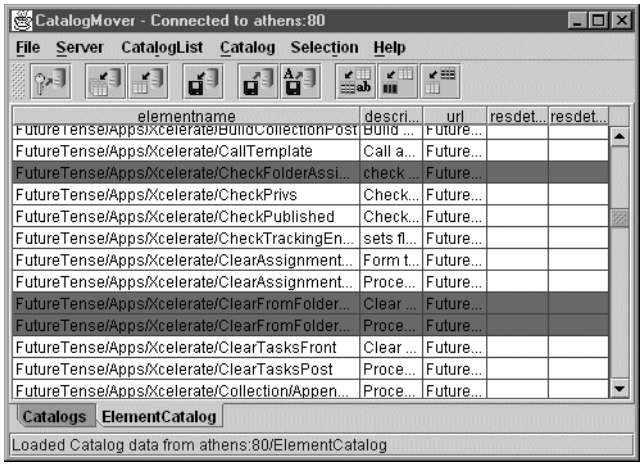
1. Choose **Selection > Select Rows By SubString**.

   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

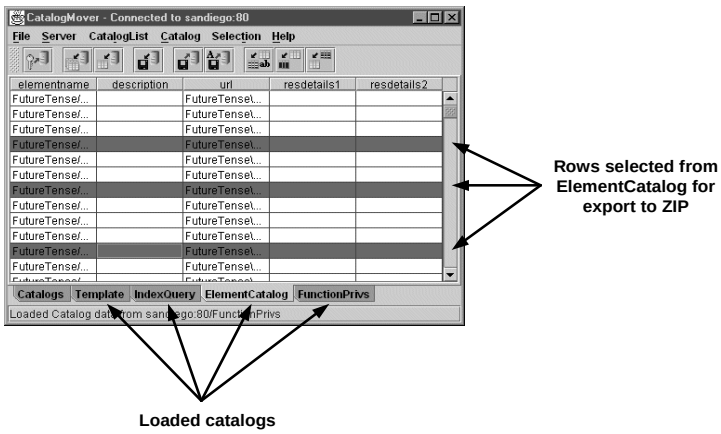
Selecting rows by substring only works for the left-most column in the table. However, you can change column positions so that any column can become the left-most column. To do this, simply click and drag the column header.

Exporting to a ZIP File

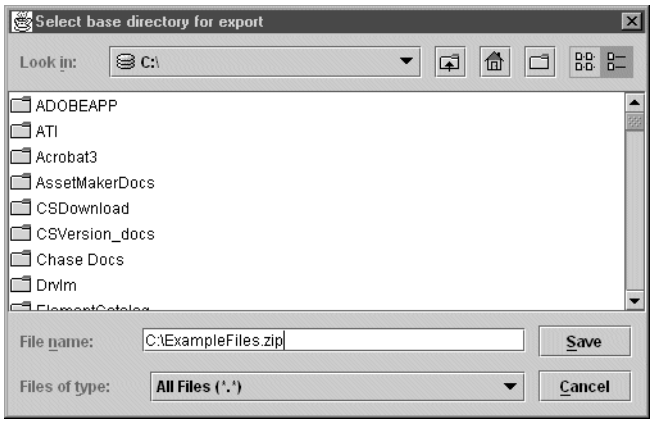
You can select several rows from several tables and export them to a ZIP file on local machine from which you are running CatalogMover. Once you create the ZIP file, you can import the contents of the file into server tables.

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:



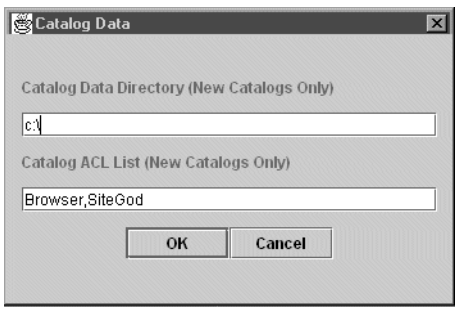
4. Navigate to the directory where you want to save the ZIP file.
5. In the **File Name** field, enter a name for the files and type a .ZIP file extension.
6. Click **Save**. The rows you selected from all of the tables are exported to a ZIP file in the directory you chose.

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 142.
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 142.
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 142.
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.

Parameters	Description
`-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
`-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.

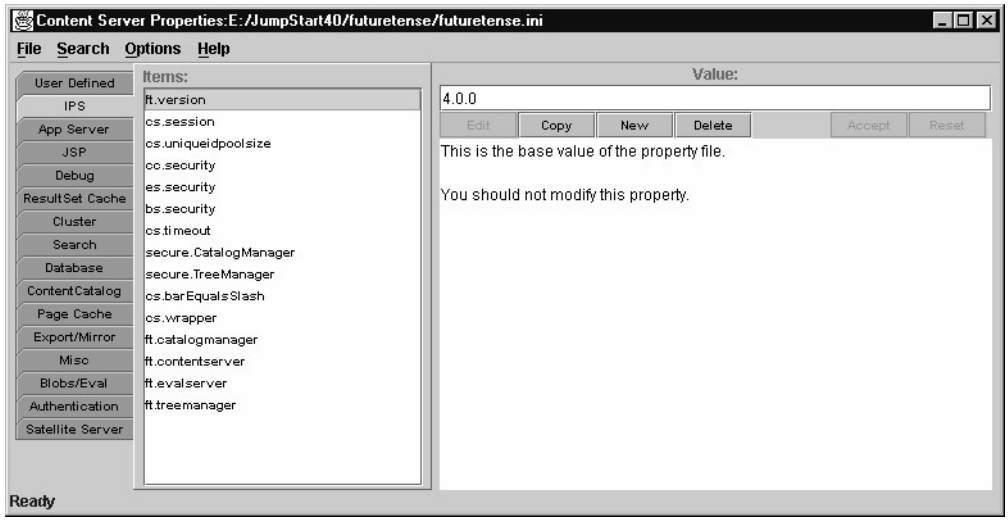
Starting the Property Editor

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.

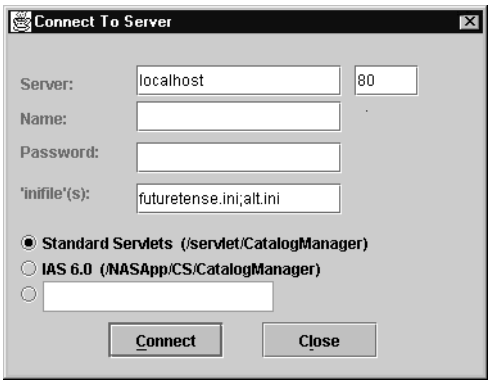
3. Select a properties group from the tabs on the left side of the window. The Property Editor displays the properties in the **Items:** pane.
4. Select a property in the **Items:** pane. The Property Editor displays the current property value and a brief description in the **Values:** pane.
5. In the **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. For example, if the `futuretense.ini` file on the server you are connecting to contains properties from another application like CS-Direct, those CS-Direct properties can be automatically added to your `futuretense.ini` file upon connection. 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`:



As another example, if you have CS-Direct installed on top of Content Server, you would merge property files for Content Server (`futuretense.ini`) and CS-Direct (`futuretense_xcel.ini`). In this case, you would enter the following:

```
futuretense.ini;futuretense_xcel.ini
```

Page Debugger

The Page Debugger is a tool that lets you step through the execution of XML and JSP elements. You can view the values of variables as the element executes, and more easily determine where errors occur in your code.

The Page Debugger provides basic debugging commands that you can use to:

- Step into a called element
- Step over a called element
- Step out of a called element
- Continue to the cursor location
- Continue executing to the end of the element file

For more information, see Chapter 9, “Error Logging and Debugging.”

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 CS-Direct and CS-Direct Advantage. For information about using XMLPost to import assets, see the following chapters:

- Chapter 17, “Importing Assets of Any Type”
- Chapter 18, “Importing Flex Assets”

Chapter 8

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:

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.

Attribute	Value
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.

Note

Set the xmldebug property to NO before running an element that calls REMOVECOOKIE. If xmldebug is set to YES, REMOVECOOKIE does not work properly.

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.

Parameter	Value
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 9

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
- Debugging Properties
- Using Error Codes with Tags
- Using the Page Debugger
- Debugging Content Server Applications

Overview

Content Server can write information related to requests it receives to the log file `futuretense.txt`, typically located in the application server folder. For example:

ContentServerInstallDir`/futuretense.txt`

The log file `futuretense.txt` contains various logged messages and errors, as well as XML parse and flow information for Content Server sessions. `futuretense.txt` includes the following information:

- XML parse and runtime errors
- Element processing messages
- Conditional evaluation prints
- Variables and their values

To enable logging, start the Property Editor and set the `ft.debug` property in the `futuretense.ini` file to yes:

```
ft.debug=yes
```

To write your own error messages to the Content Server log file, use the `ics.LogMsg` Java method. The following sample code writes a message to the logfile if the `ft.debug` property is set to `true`.

```
// ics.LogMsg() will write to the log file whether or not ftdebug
is set to true unless you wrap it in a statement like

boolean bDebug = ftMessage.truestr(ics.GetProperty("ft.debug");
if (bDebug)
        ics.logmsg("Hello");
```

By default, `ics.LogMsg` writes messages to the Content Server log file. If you want to log something to the `stdout`, use the `System.out.println` method.

You can set the maximum size (in bytes) of the log file. In a development environment, a large value is recommended.

For example, the following means the log file size will not exceed 100k bytes:

```
ft.logsize=100000
```

When the log file maximum size is reached, Content Server truncates the log file, resumes logging, and overwrites existing log file data.

See Chapter 7, “Content Server Tools and Utilities,” for information about starting the Property Editor.

Which debugging messages appear in the logfile are governed by the various debug properties. See “Debugging Properties” on page 170 for descriptions of these `futuretense.ini` properties.

Note

FatWire recommends that you enable error logging on your development or management system, but not on the delivery system. There is a significant performance setback, and the log file contains information that should not be available publicly.

Error Log File Contents

Depending on which debug properties are set, the Content Server applications all append different messages to the `futuretense.txt` file.

Error Logging (ft.debug=yes)

With debugging enabled (`ft.debug=yes`), Content Server logs error messages in `futuretense.txt` about XML and JSP tags. The messages are summarized here:

These lines indicate a request to Content Server, and report its version number:

```
Open Market, Inc. ContentServer 4.0.0
Copyright © 1999, 2000, 2001 Open Market,Inc. All Rights
Reserved.
Beta Kit Build
```

This line reports the component of Content Server and its build date:

```
CacheServer JumpStart 4.0 Build 43 Date: Nov  9 2001 at
09:13:57
```

This line reports the date/time the request was made:

```
Thu Dec 06 14:18:08 EST 2001
```

HTTP Headers

These lines report incoming HTTP headers:

```
skipping non string data for [GXAgentHops]
skipping non string data for [GX_reqstart]
skipping non string data for [GXPort]
skipping non string data for [GX_exec]
adding variable[HTTP_HOST]
adding variable[PATH_INFO]
adding variable[pagename]
adding variable[SERVER_PROTOCOL]
skipping data for [gx_session_id_FutureTenseContentServer]
skipping non string data for [GXUpdateTime]
adding variable[HTTP_CONNECTION]
adding variable[REQUEST_METHOD]
skipping non string data for [GXHost]
adding variable[REMOTE_ADDR]
adding variable[HTTP_ACCEPT]
adding variable[HTTP_USER_AGENT]
skipping non string data for [GX_stream]
logging for AppLogic+FTContentServer
```

Variable Values

These lines report default variables created and their values:

```
key:SystemAssetsRoot value:/futuretense_cs/
key:HTTP_USER_AGENT value:Java1.3.0
key:SERVER_PROTOCOL value:HTTP/1.1
key:HTTP_CONNECTION value:keep-alive
key:REQUEST_METHOD value:GET
key:QUERY_STRING value:inifile=futuretense.ini
key:SERVER_NAME value:localhost
```

```
key:REMOTE_HOST value:localhost
key:REMOTE_ADDR value:127.0.0.1
key:SERVER_PORT value:7001
key:HTTP_ACCEPT value:text/html, image/gif, image/jpeg, *;
q=.2,
      */*; q=.2
key:PATH_INFO value:/servlet/CacheServer
key:errdetail value:0
key:HTTP_HOST value:localhost:7001
key:Browser value:Unknown  Browser
key:inifile value:futuretense.ini
key:errno value:0
key:HTTPS value:off
```

Element Processing Messages

These lines indicate a call to an element. Subsequent lines are messages about processing the instructions of this element:

```
Seeding from element catalog the element:
         OpenMarket/Demos/CatalogCentre/GE/Templates/blurb-
story
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>Evaluating:
         OpenMarket/Demos/CatalogCentre/GE/Templates/blurb-
story
```

These lines indicate the results of instructions for setting variables, running a query, and doing a conditional test:

```
setting tablename to :t_images
setting id to :892041205000
no orderBy clause, ok.
[false]=[false] is true
```

This line indicates that evaluation is complete for an element:

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<Done:
         OpenMarket/Demos/CatalogCentre/GE/Templates/blurb-
story.xml
```

XML Messages (ft.xmldebug=yes)

With XML debugging enabled (ft.xmldebug=yes), Content Server writes the XML tree structure of processed pages to the log file `futuretense.txt`. For example:

```
Parsed:
DOCUMENT
|---XMLDECL
|   +---CDATA " VERSION="1.0" "
|---WHITESPACE 0xa
|---DOCTYPE  NAME="FTCS" URL="file:///E:/APPS/
futuretense_cs.dtd"
|---WHITESPACE 0xa 0xa
|---ELEMENT FTCS  Version="1.0"
|   |---ELEMENT CSVAR  NAME="SessionVariables.myvar"
|   |---WHITESPACE 0xa
|   |---ELEMENT IF  COND="IsSessionVariable.myvar=false"
```

```
|   |   |---ELEMENT THEN
|   |   |   |---ELEMENT SETSSVAR  NAME="myvar" VALUE="5"
|   |   |   +---WHITESPACE 0xa 0x20 0x20
|   |   +---WHITESPACE 0xa
|   |---WHITESPACE 0xa 0x20 0x20 0x20 0x20
|   |---ELEMENT SETSSVAR  NAME="myfive" VALUE="5"
|   +---WHITESPACE 0xa
+---WHITESPACE 0xa
```

Session Messages (ft.ssdebug=yes)

With session debugging enabled (ft.ssdebug=yes), Content Server writes the values of session variables to the log file `futuretense.txt`:

```
adding ssvariable[username] value[DefaultReader]
adding ssvariable[currentACL] value[Browser]
adding ssvariable[currentUser] value[id:DefaultReader]
```

Time Messages (ft.timedebug=yes)

With time debugging enabled (ft.timedebug=yes), Content Server writes request performance data in the log file `futuretense.txt`.

This line indicates total time related to display of a page:

```
Execute time Hours: 0 Minutes: 0 Seconds: 2:633
```

These lines indicate processing time related to queries:

```
Executed 12 Queries in Hours: 0 Minutes: 0 Seconds: 0:500
Fetched 2 Result Sets in Hours: 0 Minutes: 0 Seconds: 0:020
```

These lines indicate processing time related to parsing elements:

```
XML Parsed 1 Templates in Hours: 0 Minutes: 0 Seconds: 1:612
XML engine ran 1 templates in Hours: 0 Minutes: 0 Seconds:
1:962
```

Resultset Cache Messages (ft.cachedebug=yes)

With cache debugging enabled (`ft.cachedebug=yes`), Content Server determines when resultsets are fetched from cache and when caches are flushed, and writes this information to the log file `futuretense.txt`:

```
Table from hash...t_images

Fetched from resultset (t_images)
    key:futuretense.ini-t_images-id = '892050387000'-*

Query imgQ clearing FRS due flush
Query imgQ FRS COM.FutureTense.Common.FResultSet@1f366b
```

Page Cache Messages (ft.pgcachedebug=yes)

Controls whether Content Server should put page/pagelet caching management status messages in the `futuretense.txt` log file.

```
Error: ObjectDispatcher.Load() invalid object identifier
Variables.p
```

```
Error: ObjectDispatcher.Load() invalid object identifier
Variables.p
Parameter(s) not PageCriteria BurlingtonFinancial/AdvCols/
RecListBox
INVALID PARAM: cid=991330858149
Error: java.util.ConcurrentModificationException invoking
      method SatellitePage
```

Synchronization Messages (ft.syncdebug=yes)

Define whether Content Server logs data cache synchronization processing in the `futuretense.txt` log file.

```
SiteCatalog cache timeout absolute:false
SiteCatalog syncing cache:true
SiteCatalog cache timeout absolute:false
SiteCatalog syncing cache:true
Recording item 968685129142
Warning missing WHAT converted to * for catalog SitePlanTree
SystemEvents cache timeout absolute:false
SystemEvents syncing cache:true
Grabbing SystemEvents
```

Browser-Based Logging (ft.dbl=yes)

With browser-based logging enabled (`ft.dbl=yes`), Content Server logs messages for a specific IP address in a log file named `futuretense_`*`client_ip_address`*`.txt` in the application server installation directory.

With browser-based logging disabled (`ft.dbl=no`), all debug messages go into the log file `futuretense.txt`.

When many users are developing on a single system, this feature isolates the requests from a single client machine, thus simplifying the process of tracking information associated with a request.

You can easily display the log for a specific IP address in a browser by using the CatalogManager command, `exportlog`.

To view the log file from a browser

1. Use the Property Editor to set the `ft.dbl` property to `yes`.
2. Use your browser to go to the following URL (on the iPlanet Application Server):

   `http://`*`host`*`:`*`port`*`/NASApp/cs/CatalogManager?ftcmd=exportlog`

   or to the following URL (on a servlet engine, including Sun ONE Application Server):

   `http://`*`host`*`:`*`port`*`/servlet/CatalogManager?ftcmd=exportlog`

Additional Error Message Locations

Java run-time output provides messages and database debug messages, including:

- Stack traces for unhandled exceptions.
- Database SQL statements and errors.

Some XML tag exceptions can appear in your browser's page source window. Misspelled or erroneous tag names appear in the browser source window as text. Normal tags are processed by Content Server and replaced by generated HTML.

You can also add print statements to your source code, as shown below:

- XML code: `<CSVAR NAME="..."/>`
- Java code: `System.out.println("...");`

`Variables.errno` and `Variables.errdetail` can contain useful information. For more about `Variables.errno`, see "Using Error Codes with Tags" on page 171.

You can also find helpful information in the following additional locations:

Browser Error Messages

The browser window displays error messages, which unfortunately can be somewhat cryptic. You can also use your browser to view source code, but remember that the browser shows the generated HTML, not the original XML.

Application Server Log Files

Read the documentation for your application server to determine the location of any application server log files.

XML Syntax and Runtime Error Checking

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. However, the XML parser does not report misspelled tag names as errors, because not all tag names are required to exist in the DTD.

When a page request is made to Content Server and an XML syntax error is detected, the results streamed back contain information that can help you locate the problem. A general error description is given, followed by the offending line or column location:

This error reports a bad parameter name:

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

This 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, where the XML tags are syntactically correct, but some structural error is detected during processing. For example, this 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
```

Debugging Properties

To debug Content Server source code, set the properties in the `futuretense.ini` file, as shown in the following example. These are a few of the most commonly used properties for Content Server debugging. Your own property settings depend on your specific debugging requirements.

```
ft.debug=yes
ft.dbdebug=yes
ft.xmldebug=yes
ft.logsize=100000
cs.timeout=30000
```

The following table describes all debugging properties in the `futuretense.ini` file. (All of these properties are on the **Debug** tab in the Property Editor except for `thecs.timeout`, which is on the **IPS** tab.)

Property	Value	Description
`ft.debug`	yes | no (default)	If yes, Content Server adds debug messages to the log file `futuretense.txt`
`ft.dbdebug`	yes | no (default)	If yes, Content Server adds database debug messages to `futuretense.txt` and also to the standard output log file.
`ft.xmldebug`	yes | no (default)	If yes, Content Server adds XML evaluation messages to `futuretense.txt`
`ft.logsize`	n (in bytes; default is 10000)	Specifies the maximum size, in bytes, of the output log. FatWire recommends a large value (for example, `100000`).
`cs.timeout`	n (in seconds; default is 300)	Specifies the maximum connection idle time before Content Server terminates the connection. FatWire recommends a large value (for example, `30000`) so you can maintain context while debugging.
`ft.ssdebug`	yes | no (default)	If yes, Content Server adds session-specific debug messages to the log file `futuretense.txt`.
`ft.timedebug`	yes | no (default)	If yes, Content Server adds messages about evaluation timing to the log file `futuretense.txt`.
`ft.evaldebug`	yes | no (default)	If yes, Content Server includes diagnostic messages about EvalServer service to the log file `futuretense.txt`.
`ft.cachedebug`	yes | no (default)	If yes, Content Server adds messages about cache management status to the log file `futuretense.txt`.

Property	Value	Description
`ft.pgcachedebug`	yes | no (default)	If yes, Content Server adds page and pagelet caching management status messages to the log file `futuretense.txt`.
`ft.debugport`	n (default is 1025)	The port that the debug server uses to communicate with the template debugger.
`ft.syncdebug`	yes | no (default)	If yes, Content Server logs datacache synchronization processing.
`ft.eventdebug`	yes | no (default)	If yes, Content Server logs event management processing.
`ft.dbl`	yes | no (default)	If yes, enables browser-based retrieval of a log file with debug messages. The file `futuretense_client_ip_address.txt` is created in the application server installation directory. If no, browser-based retrieval is disabled, and all debug messages go into the log file `futuretense.txt`.
`verity.debug`	yes | no (default)	If yes, the Verity search engine adds debug messages to the log file `futuretense.txt`.

For information about all other property settings in the `futuretense.ini` file, see the *Content Server Property Files Reference*.

Note

Because enabling any debugging property can affect performance, you should not set any of the debug properties if you are doing performance testing.

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

Using the Page Debugger

The Page Debugger is a Content Server tool that lets you step through the execution of XML and JSP elements. You can view the values of variables as the element executes, and more easily determine where errors occur in your code.

The Page Debugger is installed when you install Content Server using the **Single Server with Page Debugger** option. If you do not have the Page Debugger, you can install it by upgrading your existing installation. To do so, run the Content Server upgrade script and select the **Upgrade with Page Debugger** option.

FatWire recommends that you install the Page Debugger on your development system only.

Invoking the Page Debugger

Invoking the Page Debugger is a two-step process:

1. Start the Debug Listener.
2. Alter the URL that invokes a page.

Start the Debug Listener

Before running the Page Debugger, you must first start the **Debug Listener**.

To start the Debug Listener

1. Run the `DebugListener.bat` file.

   The following Debug Listener window appears:

2. By default, the Debug Listener runs on port 1025. If you want to run the Debug Listener on another port (for example, because another service is already using port 1025), then do both of the following:
   - Use the Property Editor to change the `ft.debugport` setting to your chosen port number; for example, `2025`.
   - On the Debug Listener invocation line, use the `-p` option, followed by a space and then the port number. For example:

     ```
     java -classpath cs.jar
     COM.FutureTense.Apps.DebugListener -p 2025
     ```

Alter the URL that Invokes a Page

With Debug Listener running, use your Web browser to request the page to be debugged. In the browser's **address** field, replace the phrase `ContentServer` with `DebugServer`. The page is then requested through the Page Debugger instead of Content Server.

Note

The Debug Listener and the browser that you use to make the page request to the DebugServer must be on the same machine.

If you are running iPlanet Application Server, replace the following address:

`http://host:port/NASApp/CS/ContentServer?pagename=xxx`

with this address:

`http://host:port/NASApp/CS/DebugServer?pagename=xxx`

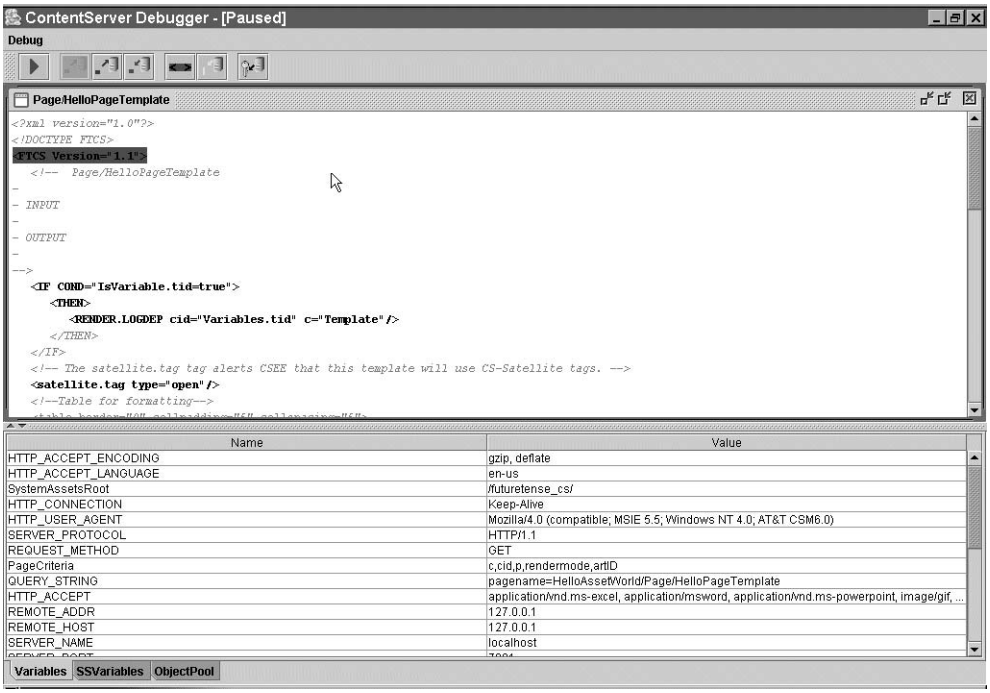
If you are running WebLogic or WebSphere, replace the following address:

`http://host:port/servlet/ContentServer?pagename=xxx`

with this address:

`http://host:port/servlet/DebugServer?pagename=xxx`

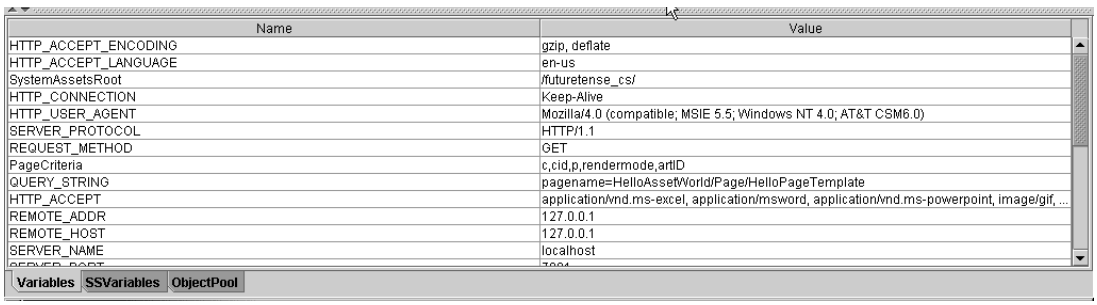
The Debugger main window appears:



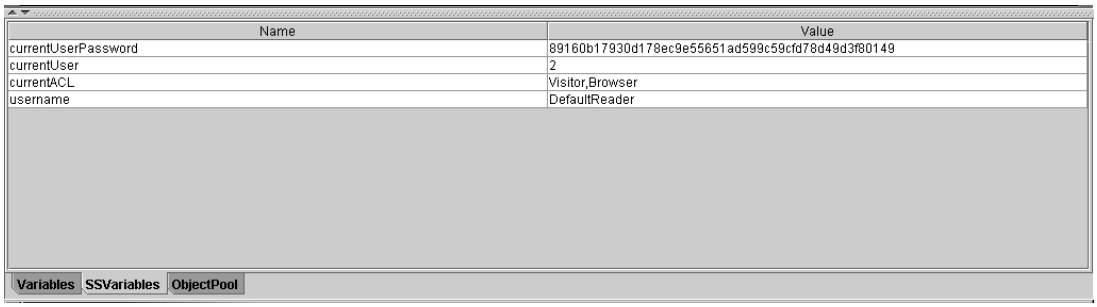
The tabs in the bottom pane of Page Debugger's window provide the following information:

- Names of variables and their values
- The names and values of objects in the object pool
- The state of all session variables and their values

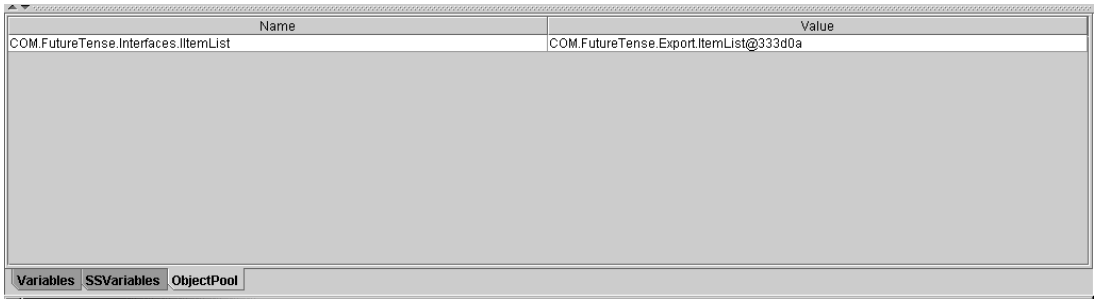
In the following examples, the Page Debugger windows display information about session variables, objects, and variables:



The **Variables** window displays the names and values of the variables that are used in the element that you are currently debugging.



The **SSVariable** window shows the names and values of all the session variables in the element that you are currently debugging.



The **ObjectPool** window shows the names and values of all of the objects in the object pool.

Page Debugger Commands

The Page Debugger has the following commands:

- Continue to Next Breakpoint
- Step Into
- Step Over
- Step Out
- Toggle Breakpoint
- Continue to Cursor
- Done

To choose a command, click the buttons in the Page Debugger button bar, or choose any of the following commands from the **Debug** menu:

Continue to Next Breakpoint

Use **Continue to Next Breakpoint** to continue to the next breakpoint that you have set using the **Toggle Breakpoint** button.

Note that you cannot set breakpoints until you start a debugging session; as a result, you cannot use the **Continue to Next Breakpoint** button to step though an element the first time you debug it.

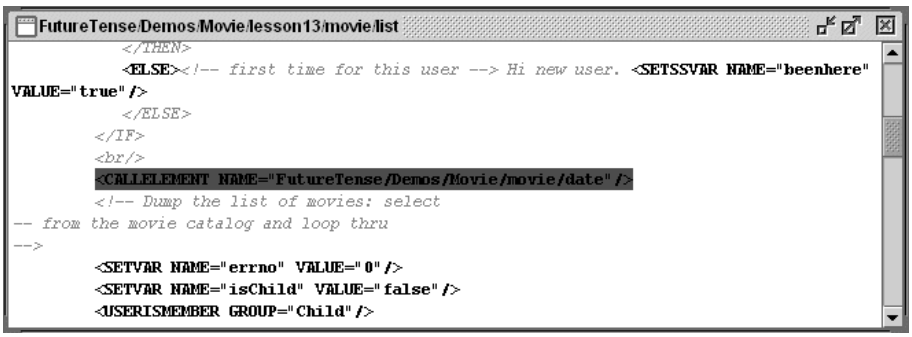
For example, the first time you debug a page, you have to step through it manually, setting breakpoints as you go. If you refresh the page to debug it again, the breakpoints you set will be there, and you can use the **Continue to Next Breakpoint** button to step though the element.

Breakpoints do not persist between sessions of the DebugListener.

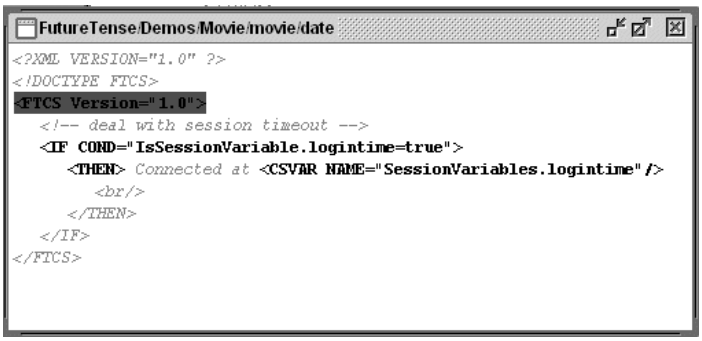
Step Into

Use **Step Into** to open the element called by the element that you are currently debugging.

To use the **Step Into** function, select the `CALLELEMENT` tag and click the **Step Into** button. In the following example, the element being called is used to display the date to a visitor who is logged in to the site:



The following window appears:



Step Over

Use **Step Over** to move from the currently highlighted tag to the next tag in the element, without executing any elements that are called in between the tags.

Step Out

Use **Step Out** to step out of the elements currently opened by the Page Debugger. If you have opened an element as a result of using the **Step Into** command, use **Step Out** to return to the original Page Debugger file from which you opened the element.

You can also use **Step Out** to exit the original element that you opened with the Page Debugger.

Toggle Breakpoint

Use the **Toggle Breakpoint** button to set or remove breakpoints in the elements you have opened in Page Debugger.

For more information about breakpoints, see the "Continue to Next Breakpoint" section of this chapter.

Continue to Cursor

Use **Continue to Cursor** to select a location within the element where you want the Page Debugger to move to next. To check a specific tag located in the middle of the element, complete the following steps:

1. Use the cursor to select part of the tag.
2. Click **Continue To Cursor**.

The Page Debugger highlights the tag that you selected with your cursor.

Note

You must select at least one text character or space in the tag for **Continue To Cursor** to work. It does not work if you simply place the cursor within the tag.

Done

Use **Done** to complete the Page Debugger session and exit the file. When you click **Done**, the Page Debugger moves through the entire file, listing all information in the bottom window of the Page Debugger main view.

Debugging Content Server Applications

This section provides some debugging and error logging information for developers using other Content Server products. It contains the following sections:

- Debugging Engage
- Property Messages
-

Debugging Engage

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, or by examining it with the Page Debugger utility.

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 and examine values stored for specific visitor attributes, history attributes, and history types (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 type 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 type.

- `<VDM.GETHISTORYEARLIEST VARNAME="varname" [STARTDATE="date1" ENDDATE="date2" LIST="constraints"] />` retrieves the timestamp of the first time the specified history type 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 type 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.

Verifying Visitor Data Assets

To determine that you correctly set up your visitor attributes, history attributes, and history types, 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 types that you created.
- Determine that the constraint types are correct and that the input ranges are accepting the correct range of input.

For help with creating segments, see the *Content Server User's Guide*.

Verifying Recommendation Assets

To verify that you configured your recommendation assets correctly, complete the following kinds of exercises:

- Create some test segments.
- In the product and product group forms, assign ratings for the segments.
- Browse your site as a visitor and register yourself so that you qualify for the test segment.
- Examine the items that the recommendation assets return.
- If you find problems, use the Engage XML or JSP object methods to write test pages that isolate the problem.

Property Messages

Message Text	Meaning
`Cache max error`	Error loading the `Caching.cache_max` property.
`Cache check time error`	Error loading the `Caching.cache_check_interval` property.
`Time pattern error`	Error loading the `Caching.expiration` property.
`Read timeout error`	Error loading the `Configuration.readtimeout` property.
`Block timeout error`	Error loading the `Configuration.blocktimeout` property.
`Refresh time error`	Error loading the `Configuration.control_refresh_interval` property.
`File size error`	Error loading the `Caching.file_size` property.
`Cache max size:`	The `Caching.cache_max` property value has been loaded.
`Cache check interval:`	Error loading the `Caching.cache_check_interval` property value.
`host <Host:Port>`	The `Remote Host.host` property value has been loaded.
`Default fragment expiration:`	The `Caching.expiration` property value has been loaded.
`Socket timeout value:`	The `Configuration.readtimeout` property value has been loaded.
`Thread block time:`	The `Configuration.blocktimeout` property value has been loaded.

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 10, “Data Design: The Asset Models”
- Chapter 11, “The Content Server Database”
- Chapter 12, “Managing Data in Non-Asset Tables”
- Chapter 13, “Resultset Caching and Queries”
- Chapter 14, “Designing Basic Asset Types”
- Chapter 15, “Designing Flex Asset Types”
- Chapter 16, “Designing Attribute Editors”
- Chapter 17, “Importing Assets of Any Type”
- Chapter 18, “Importing Flex Assets”
- Chapter 19, “Importing Flex Assets with the BulkLoader Utility”

Chapter 10

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
- Search Engines and the Two Asset Models
- Tags and the Two Asset Models
- Summary: Basic and Flex Asset Models

Designing and creating tables that do not hold assets is discussed in Chapter 11, “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) 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 Export to Disk publishing method. There are very limited applications of the flex asset model in which it makes sense to use the Export to Disk 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 14, "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 with the 6.1 version of the CS-Direct application 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 • Query assets • Collection assets • Template assets	• All custom basic assets (made with AssetMaker) • Article assets • Image assets • Linkset assets • Recommendation assets • Link assets • Page assets

For information about setting configurable subtypes, see Chapter 14, "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 191.)
- 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 22, "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:

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. You cannot change the value in this field.	Forms: • Inspect • Edit • Status • search forms
`name`	A unique name for the asset. Names are limited to 64 alphanumeric characters.	Forms: • New • Edit • Inspect, • Status Also in the search results lists.
`description`	A short description of the asset that offers more information than just the name.	Forms: • New • Edit • Inspect • Status Also in the search results lists.
`status`	The status of the asset, one of the following status codes obtained from the `StatusCode` table: PL - created ED - edited RF - received (from XMLPost, for example) UP - upgraded from Xcelerate 2.x VO - deleted (void) 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) Note that assets with a status of VO (deleted) are not displayed anywhere in the Content Server Windows interface.

Default Column (Field) Name	Description	Where It's Displayed in the Content Server Interface
`createdby`	The identity of the user who originally created the asset. This user name is obtained from the `SystemUsers` table. CS-Direct controls the value in this field; it cannot be edited manually.	Forms: **Status** 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. CS-Direct controls the value in this field; it cannot be edited manually.	Forms: **Status** 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. CS-Direct controls the value in this field; it cannot be edited manually.	Forms: **Status** 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. CS-Direct controls the value in this field; it cannot be edited manually.	Forms: **Status** Also, if revision tracking is enabled for assets of this type, the **Revision History** list.
`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. The promotion asset type is the only default asset type that uses this column. If you want to use the startdate and enddate fields for your asset types, see "Example: Enabling path, filename, startdate, and enddate" on page 291.	Forms: • **Duration**, **Edit**, and **Inspect** for promotion assets. • **New**, **Edit**, **Inspect**, and **Status** if you enable it for other asset types.

Default Column (Field) Name	Description	Where It's Displayed in the Content Server Interface
`enddate`	For promotion assets (a Engage asset), this column stores the end time of the promotion's duration The promotion asset type is the only default asset type that uses this column.	Forms: • **Duration**, **Edit**, and **Inspect** for promotion assets • **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 192.	Forms: • **New**, and **Edit** for Template assets (**Asset Type** field) • **New**, and **Edit** for query assets (**Result of Query** field) • **New**, and **Edit** for any asset type that has subtypes configured for it • **Set Default Templates**
`filename`	The name to use for the file created for this asset during the Export to Disk publishing method. The page and article asset types are the only asset types that have this field enabled by default. If you want to use the filename field for your asset types, see "Example: Enabling path, filename, startdate, and enddate" on page 291	Forms: • **New** and **Edit** for page and article assets, by default • **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. The page and article asset types are the only asset types that have this field enabled by default. If you want to use the filename field for your asset types, see "Example: Enabling path, filename, startdate, and enddate" on page 291.	Forms: • **New** and **Edit** for page and article assets, by default • **New** and **Edit** for any other asset type that has the field enabled

Default Column (Field) Name	Description	Where It's Displayed in the Content Server Interface
`template`	The template for the asset. 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. 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: • **New** • **Edit** • **Inspect** • **Status**
`category`	The category code of the category assigned to the asset, if any. 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: • **New** • **Edit** • **Inspect** • **Status**
`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. 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. 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

CS-Direct Advantage delivers the flex asset model. This 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.
- 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 211.
- 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.

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 2 lists the data types for flex attributes, the properties that define the data types, and the files where the properties are located:

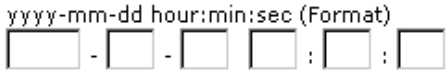
Table 2: Data types for flex attributes

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 16, “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 231.
- 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.

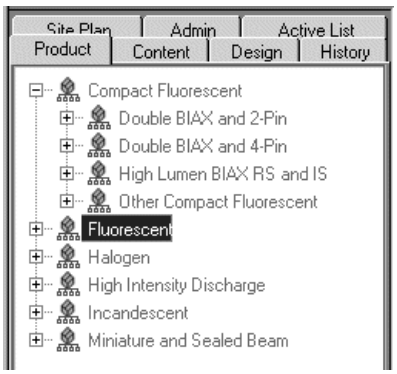
Each parent asset type has its own set of attributes, as specified in its parent definition. The parent definition creates a form that you see in the Content Server interface.

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.

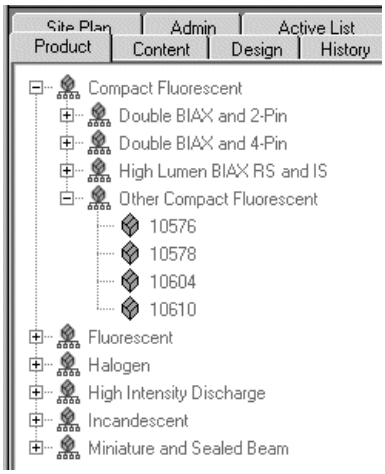
A **flex definition asset** describes one kind of flex asset in a flex family; for example, a shoe, a toaster, a bowling ball, a brochure, a newsletter, an article, and so on. A flex definition asset is a template in that it directly affects a form that you see in the Content Server interface.

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 Filters

Flex filters can transform, categorize, or perform other kinds of automated actions on the data in a flex attribute when a content asset is saved.

For example, imagine that your content providers use Microsoft Word to author their content and then use the CS-DocLink interface to save their Word documents as assets. Because you want to display their content in HTML format, you decide to configure a flex filter that converts the data in the `.doc` file to HTML and then saves and stores that data as an `.htm` file when they save the asset. You then design a template that extracts and displays the value of the attribute that holds the `.htm` file rather than the attribute that holds the `.doc` file.

There are several parts to the flex filter framework:

- Flex filter classes
- Registered transformation engines
- Flex filter assets

Flex Filter Classes

Flex filter classes implement the transformation, classification, or other action. These classes are listed in the `Filters` table in the Content Server database. When you create a filter asset, you select a flex filter class for it.

As of version 5.5.1, CS-Direct delivers one flex filter class: Document Transformation. This filter converts a document from one file type into another by invoking a registered transformation engine.

Registered Transformation Engines

Registered transformation engines are document conversion engines that are specified in the `SystemTransforms` table. If you create a filter asset that uses the Document Transformation class, you must also specify which transformation engine to use.

As of version 5.5.1, CS-Direct delivers one transformation engine, the Verity Keyview engine. This engine can convert up to 200 kinds of binary document files (.doc, .pdf, .txt,

and so on) to either HTML or XML files. By default, it is configured to convert files to HTML.

Note that other packages that you purchase from FatWire to work with your Content Server system might register a transformation engine.

Flex Filter Assets

Flex filter assets do the following two things:

- Specify which registered flex filter class to use.
- Specify which information is passed to the filter class (through arguments). For example, information about the data so the filter class knows which data to convert and where to store it when the asset is saved.

You assign flex filter assets to flex definition and flex parent definition assets.

When you create a flex filter asset, you specify the flex filter class to be used and then provide values for the arguments that the filter class needs in order to perform its action. The Documentation Transformation filter class (the default class) expects values for the following arguments:

- **Document Transformer Name** – the name of a registered transformation engine exactly as it is listed in the `SystemTransforms` table. The name of the Verity Keyview engine configured to convert binary files to HTML is listed as `Verity: Convert to HTML` in the `SystemTransforms` table.
- **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 binary file in that attribute.
- **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 binary file.

  The data stored in the output attribute (field) is read-only because it has been 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. When you specify that the document transformation engine is `Verity: Convert to HTML`, the document extension must be either `.htm` or `.html`.

After you create a flex filter asset, you assign it to the appropriate flex definition or flex parent definition assets. Then, when content providers create a flex asset of that definition, the filter performs its transformation or assessment when they save their assets.

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 195.

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. 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. 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.)
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 225), 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. 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 14, "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 192.	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`
Publishing methods	Export to Disk Mirror to Server	Export to Server is possible, but is atypical for the flex model. 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
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

Name of asset type	Product or Sample Site
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 11

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 Remote Databases

For information about managing the data in non-asset tables, see Chapter 12, “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 195.

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 199.)
- `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 13, “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 20, "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 20, "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. For information about creating ACLs, see the *Content Server Administrator's Guide*. For information about using ACLs to implement user management for your online site, Chapter 25, "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. 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 277. For information about the data types for flex attributes, see “Data Types for Attributes” on page 202.

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 the performance of selecting data degrades with large field sizes

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 226
- The upload field examples for basic asset types, starting with "Upload Example 1: A Standard Upload Field" on page 290
- The procedure for creating flex attributes of type blob in the section "Creating Flex Attributes of Type Blob (Upload Field)" on page 323.

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 14, "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 15, "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:

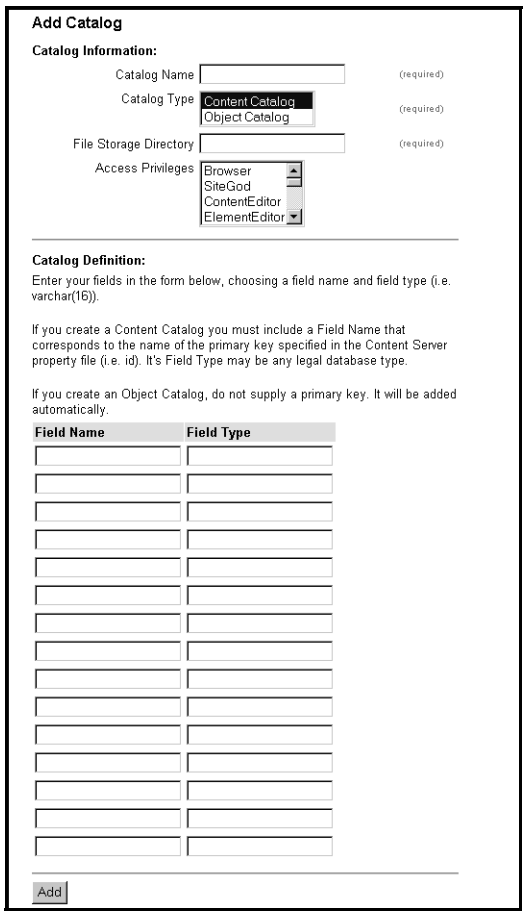
     `http://your_server/Xcelerate/LoginPage.html`

  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 223.

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 12, "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 219. 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 12, "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:

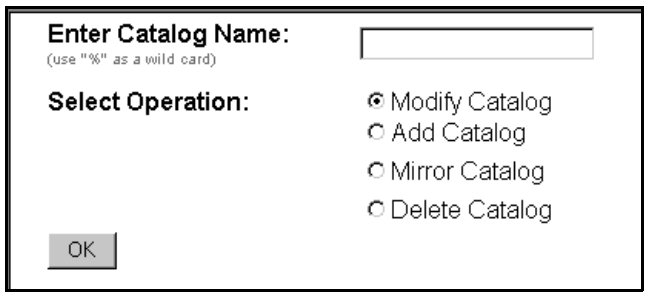
   `http://`*`your_server`*`/Xcelerate/LoginPage.html`

   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**.

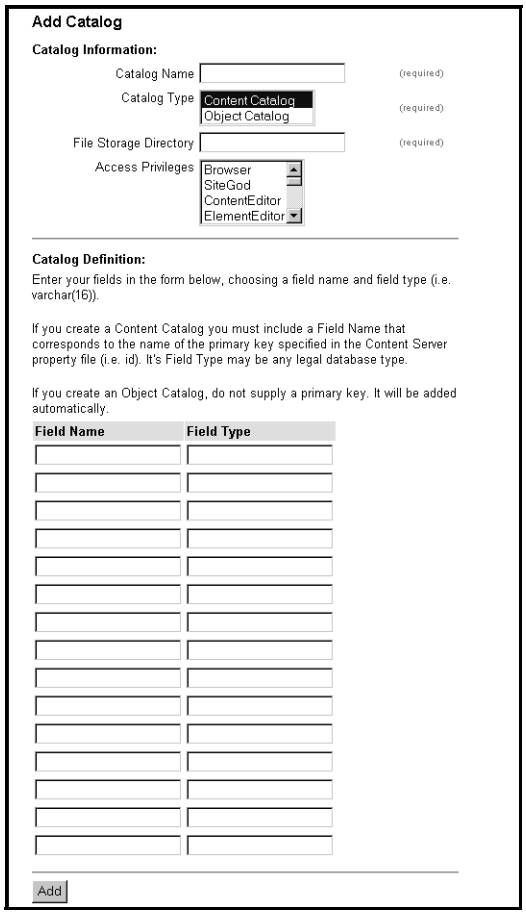
   The Content Server user interface appears.

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 12, “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 the Content Server Explorer and log in to the Content Server database.
2. Double-click on the **SystemInfo** table.

3. Right-click the mouse on 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: • When you create a Template asset, CS-Direct automatically creates a page entry for it in the `SiteCatalog` table. • When you create a SiteEntry asset, CS-Direct automatically creates a page entry for it in the `SiteCatalog` table. • You can use the Content Server Explorer tool or the “Site” form in the Content Server Management Tools interface. 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. 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: • When you create a Template asset, CS-Direct automatically creates an entry for it in the ElementCatalog table. • When you create a CSElement asset, CS-Direct automatically creates an entry for it in the ElementCatalog table. • You can use the Content Server Explorer tool to add non-asset elements. For information about coding elements and pages, see Chapter 23, "Coding Elements for Templates and CSElements."
SystemACL	The "ACL" form in the **Content Server Management Tools** node. 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. The **only exception** to this rule is if you need to identify a foreign table to Content Server.
SystemSQL	The Content Server Explorer tool. For information about the various kinds of queries that are available, see Chapter 13, "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 Remote 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.

Property Files for Remote Databases

Content Server can also access remote databases. If you want to access and work with data that is kept in a remote database, you must create or identify a property file for that database.

Any property file for a remote database must be located in the same directory as the Content Server `futuretense.ini` file.

Additionally, it must include **all** of the **database properties** that are listed in the *Content Server Property Files Reference*.

Accessing the Property File for a Remote Database

To access a remote database from a Content Server page, use the `ics.LoadProperty` Java method or the `LOADPROPERTY` XML tag to specify the property file that identifies that database before the statement of the operation that you want to perform in that database.

For example:

```
<LOADPROPERTY FILE="example.ini"/>
```

After completing the operation on that remote database, be sure to re-establish the connection with the primary Content Server database by closing the connection to the remote database with the `ics.RestoreProperty` Java method or the `RESTOREPROPERTY` XML tag.

For example:

```
<RESTOREPROPERTY CLOSE="true"/>
```

Chapter 12

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

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 17, “Importing Assets of Any Type” and Chapter 18, “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:

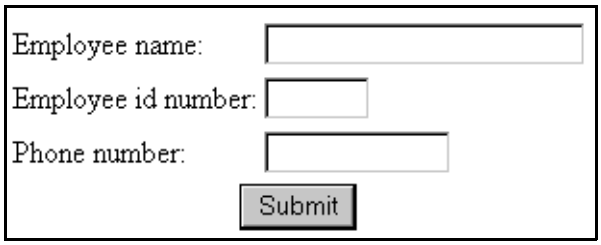
Field	Data type
id	VARCHAR(6)
phone	VARCHAR(16)
name	VARCHAR(32)

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 223

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 240 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:

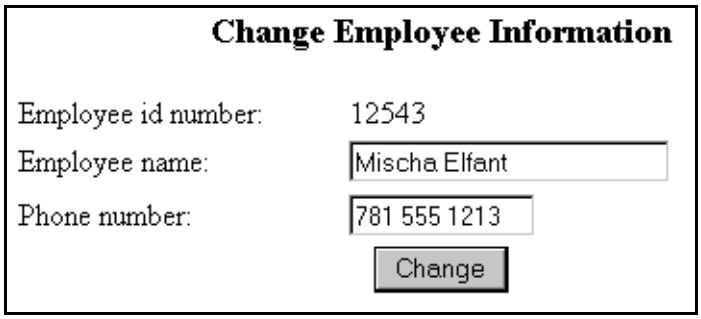
```
<?xml version="1.0" ?>
<!DOCTYPE FTCS SYSTEM "futuretense_cs.dtd">
<FTCS Version="1.1">
<!-- Documentation/CatalogManager/SelectNameForm
-->

<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 13

Resultset Caching and Queries

The CatalogManager servlet (and its API) maintains the resultset cache on your Content Server systems. This chapter describes how to enable resultset caching and how to create queries that allow CatalogManager to accurately cache resultsets and to then flush those resultsets from the cache.

You or your system administrators set up resultset caching on all three systems (development, management, and delivery).

This chapter contains the following sections:

- Overview
- How Content Server Identifies a Resultset
- Specifying the Table Name
- Flushing the Resultset Cache
- Enabling Resultset Caching
- Summary

Overview

Whenever the database is queried, the 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.

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 has not changed 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 the cache is always faster than performing another database lookup.

The resultset cache is a hash table held in Java memory. The resultsets in this hash table 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 (the `ft.synch` property in the `futuretense.ini` file).

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.

How Content Server Identifies a Resultset

The resultset cache is a hash table and the key that identifies an individual resultset for a given table name is the combination of the values of two database connection properties plus the text string of the query itself.

For example, the hash name used to identify a resultset from a table in the Content Server database is created as follows:

The value of the `cs.dsn` property from the `futuretense.ini` file + the value of the `cs.privuser` property from the `futuretense.ini` file + the actual query string.

If you query a remote database (remember that you must first use the `ics.LoadProperty` method or the `LOADPROPERTY` tag to specify the property file that identifies that remote database before the statement of the query), the hash name is created with the `cs.dns` and `cs.privuser` properties from the property file used to identify that database.

This means that if you run queries against a remote database and any of the table names are the same as a table name in the Content Server database, the hash names of the resultsets for those queries will be different even though they will be listed under the same table name in the cache (that is, the hash map is the same). This means that Content Server can flush the correct resultset when a table is updated.

If you do not load the property file for another database before running the query, Content Server assumes that it is connecting to the Content Server database.

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

The following table presents the three properties in the `futuretense.ini file` that control the resultset cache for all tables that you have not added table-specific caching properties for:

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. Setting this value to -1 disables resultset caching for all tables that do not have their own caching properties configured.
`cc.cacheResultsTimeout`	The default amount of time (number of minutes) to keep a resultset cached in memory. Setting this value to -1 means that there is no timeout value for tables that do not have their own caching properties configured.
`cc.cacheResultsAbs`	How to calculate the expiration time. If the value is set to `true`, the expiration time for a resultset is absolute. If the timeout is set to 5 minutes, then 5 minutes after it was cached, it is flushed. If the value is set to `false`, the expiration time for a resultset is based on its idle time. For example, if the timeout is set to 5 minutes, it is flushed 5 minutes after the last time it was requested rather than 5 minutes since it was originally cached.

To change these properties, open the `futuretense.ini` file with the Property Editor utility and modify them. For information about using the Property Editor utility, see Chapter 7, “Content Server Tools and Utilities.”

Table-Specific Properties

CatalogManager not only uses the properties described in the preceding table, it also checks the `futuretense.ini` file to see 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 was as do the default properties defined in the table in the preceding section. All system tables have these properties set for them.

The syntax for your custom properties is as follows:

```
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.

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 7, “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 to use is to identify a large group of similar tables that you can use the default properties for 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 a custom properties for each table in the database on that system—at the very least, 50-100 of them.

Note

If you set the `ft.cachedebug` property to `yes`, debugging messages about the resultset cache are written to the `futuretense.txt` log file.

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 14

Designing Basic Asset Types

As mentioned in Chapter 10, “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 CS-Direct 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 CS-Direct 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 272 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 it on the sites that need to use it, 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

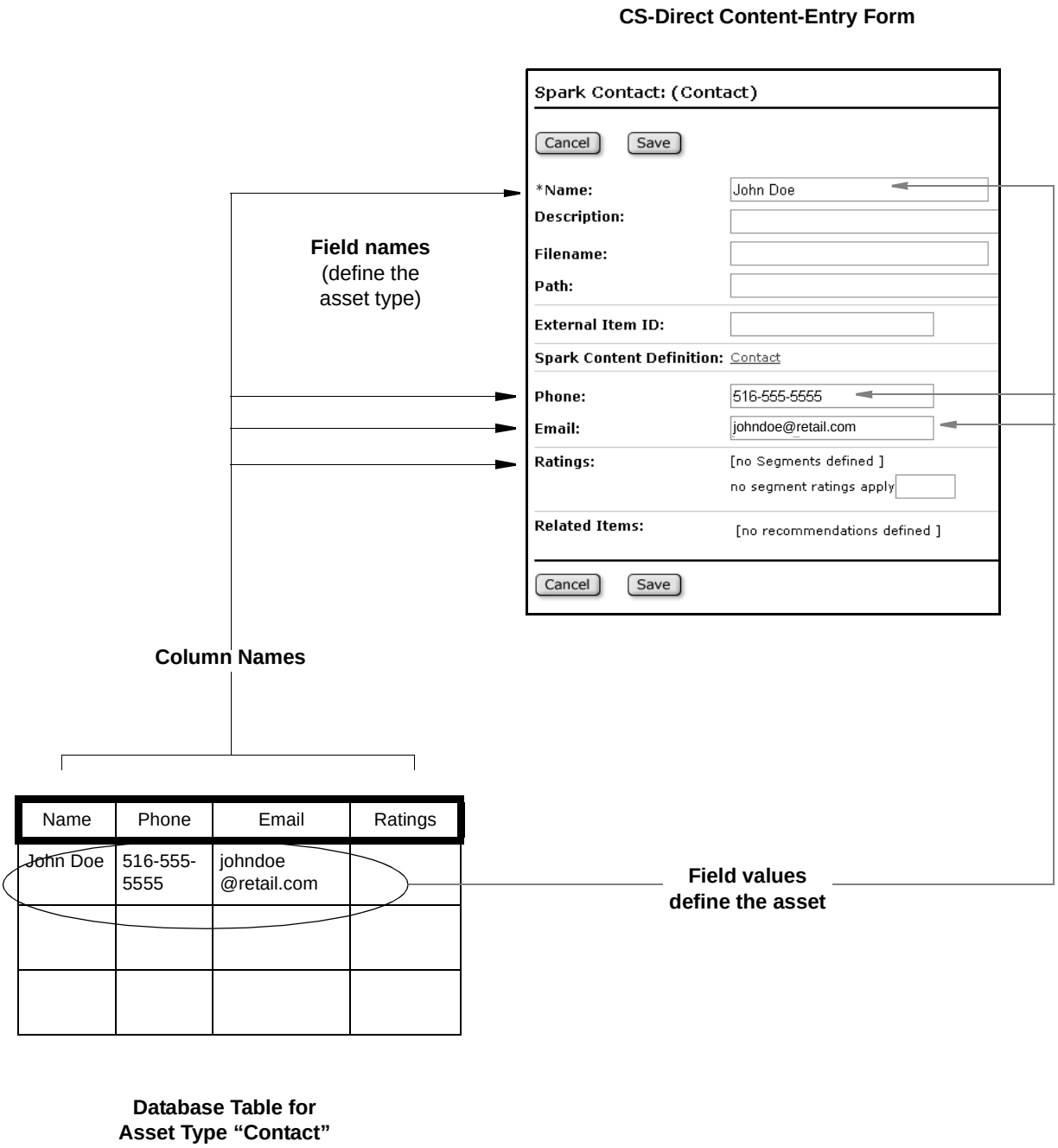
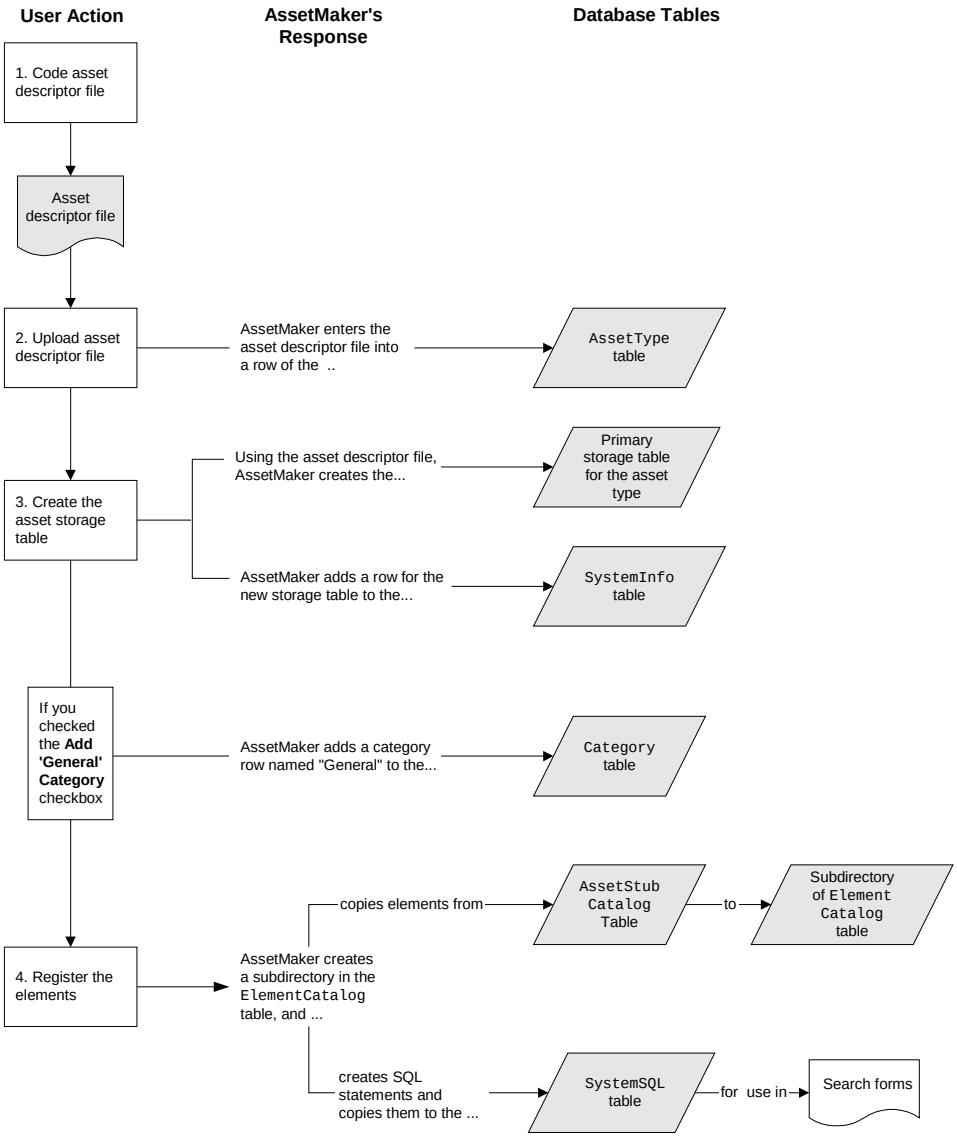


Figure 5: How AssetMaker Works



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. The closing <\PROPERTIES> tag There is only one PROPERTIES tag pair in each asset descriptor file.

  Note that 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 name of the column that will hold the values entered for this property, and the name of the field on the form that will be rendered for users 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 the asset type, it creates columns for all the properties defined by the pairs of PROPERTY tags in the asset descriptor file.

However, CS-Direct needs several default columns for its basic functionality and so AssetMaker creates each of the default columns in the asset type's storage table in addition to the columns defined in the asset descriptor file for that asset type.

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 195.

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 289.

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. Corresponds to the HTML input type named `TEXT`.

Input TYPE	Description
TEXTAREA	A text box, with scroll bars, that accepts multiple lines of text. Corresponds to the HTML input type named TEXTAREA. 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. 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. Note that 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.
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. 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. 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. 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. Note: the size of a file that is selected in an upload field cannot exceed 30 megabytes.
SELECT	A field that presents a drop-down list of options that can be selected. 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. Corresponds to the HTML input type SELECT.
CHECKBOX	A check box field. 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. Corresponds to the HTML input type CHECKBOX.

Input TYPE	Description
RADIO	A radio button control. 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. 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. 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.
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: • 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* Where *myAssetType* is the asset type that you are creating the custom field for, and *fieldname* is the name of the custom field. An ELEMENT field can have any storage type, including BLOB.

Datatypes for Standard Asset Fields

You can customize the appearance of CS-Direct's standard asset fields, however, the datatypes of these fields must not be changed.

System fields, which are identified in the following table, can be altered cosmetically, but their behavior cannot change. For other fields, the length of the varchar can be changed, but the datatype must remain the same. The following table lists the datatypes for standard fields:

Field	Datatype
ID(System Field)	NOT NULL NUMBER(38)
NAME	NOT NULL VARCHAR(64)
DESCRIPTION	VARCHAR(128)
TEMPLATE(System Field)	VARCHAR(64)
SUBTYPE	VARCHAR(24)
FILENAME	VARCHAR(64)
PATH	VARCHAR(255)
STATUS(System Field)	NOT NULL VARCHAR(2)
EXTERNALDOCTYPE (System Field)	VARCHAR(64)
URLEXTERNALDOCXML(System Field)	VARCHAR(255)
URLEXTERNALDOC(System Field)	VARCHAR2(255)
CREATEDBY(System Field)	NOT NULL VARCHAR(64)
UPDATEDBY(System Field)	NOT NULL VARCHAR(64)
CREATEDDATE(System Field)	NOT NULL DATE
UPDATEDDATE(System Field)	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 8: (Optional) Customize the Asset Type Elements” on page 298.

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. 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. 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. 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. 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. 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 a search engine such as Verity or AltaVista. 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 8: (Optional) Customize the Asset Type Elements" on page 298.
IndexReplace	See the description of IndexAdd, above.
IndexCreateVerity	If you are using Verity as your search engine, this element interacts with the IndexAdd and IndexReplace elements to establish which fields to index.
Tile	Formats the **Search Results** page, a page that lists the assets that meet the search criteria, for assets of this type. When the page is rendered, AssetMaker uses the SEARCHRESULTS statements in the asset descriptor file to display the results.
LoadTree	Determines how the 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. 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. 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.
2. Use AssetMaker forms (accessable from the **Admin** tab in the Content Server user interface) to upload the asset descriptor file, create the database table, and copy the asset type elements from the `AssetStubElementCatalog` table to the appropriate directory in the `ElementCatalog` table.
3. Configure the asset type.
4. 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.
5. 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.
6. (Optional) If necessary, customize the asset type elements.
7. (Optional) Create asset **Association** fields for the new asset type.
8. (Optional) Add **Category** entries for the new asset type.
9. (Optional) Add **Source** entries for the new asset type.
10. (Optional) Add **Subtype** entries for the new asset type.
11. (Optional) Add **Mimetypes** for the new asset type.
12. (Optional) If you are using a search engine (Verity or AltaVista) rather than the CS-Direct 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.
13. Code templates for assets of this type. (See Chapter 23, “Coding Elements for Templates and CSElements.”)
14. Move the asset types to the other systems, management and delivery. 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 of the CS-Direct features will you use to organize or categorize assets of this type? For example, source, category, and asset associations. For each one, determine their names 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 that 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 274, 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 any of the standard asset fields by including them in your asset descriptor file. You may not change the storage type of a standard asset field. For a list of these storage types, see “Datatypes for Standard Asset Fields” on page 280.

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 225.

- 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 195.)
- 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. It is not delivered with CS-Direct or CS-Direct Advantage. For information about configuring your system to use eWebEditPro, see the *Content Server Administrator's Guide*.

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 exerpt 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 3: Upload the Asset Descriptor File in 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:

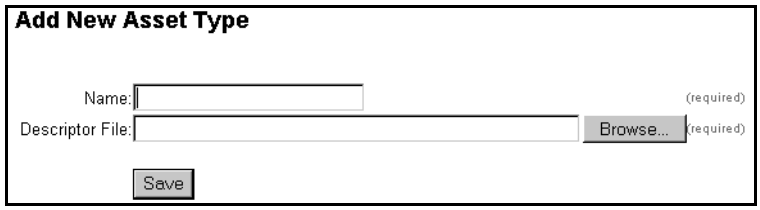
1. Open your browser and enter this address:

   `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.

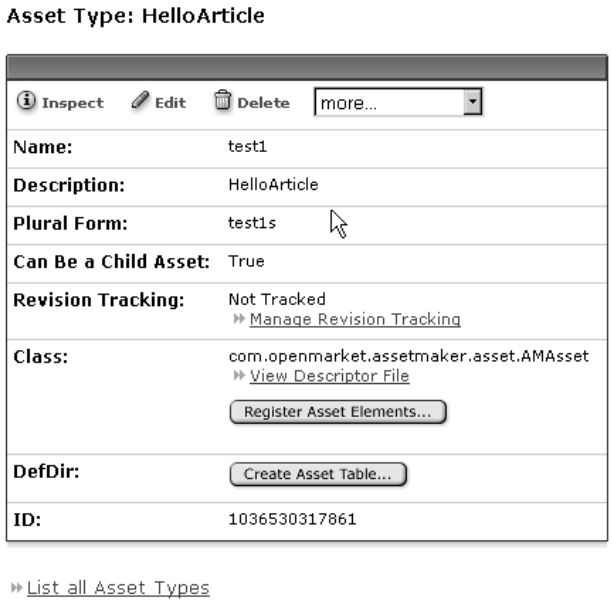
3. Select the **Admin tab** > **AssetMaker** > **Add New**.

   The “Add New Asset Type” form appears:



4. 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.
5. Click the **Browse** button next to the **Descriptor File** field and select the asset descriptor file.
6. 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 4: Create the Asset Table (continued from Step 3)

7. Select **Create Asset Table**.
8. 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.

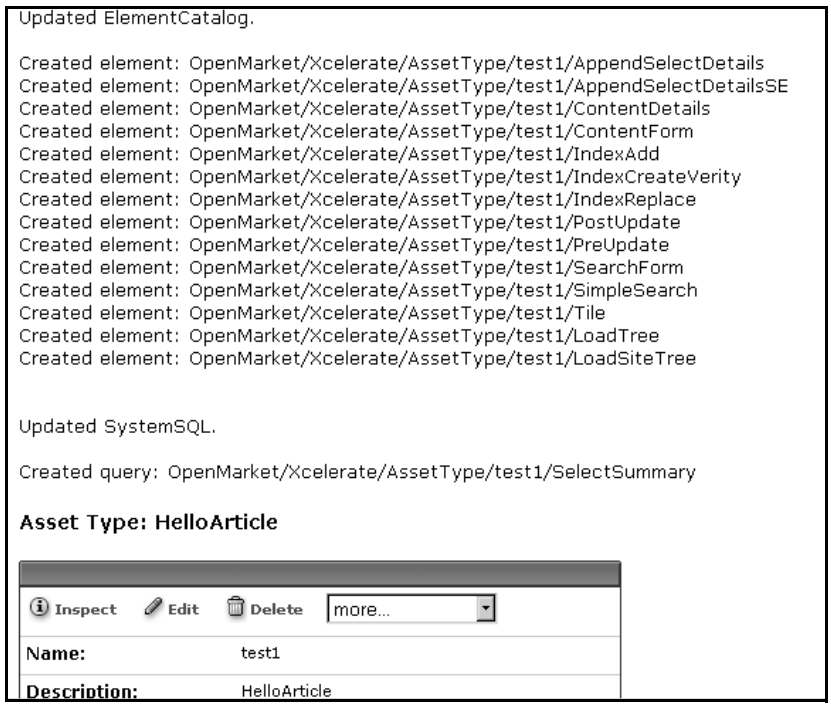
If you check the **Add 'General' category** checkbox, AssetMaker adds one row to the `Category` table for the new asset type and names that category **General**.

9. Click **Create Asset Table**.

   AssetMaker creates the table.

10. 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 summary like this one:



Step 5: 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 6: 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 7: 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 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 3: Upload the Asset Descriptor File in to Content Server" on page 294.
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 8: (Optional) Customize the Asset Type Elements

There are two ways to customize your asset type elements:

- If your management system requires the same modifications for assets of all types, modify the source elements in the `AssetStubElementCatalog` table before you create your asset types. That way you only have to make your customizations once.
- If your change is specific to a certain asset type, modify the elements in that asset type's directory in the `ElementCatalog` table.

If you change the source elements (stub), you must re-register all of the asset types that you want to take the new changes.

Note that, although customizing elements might be necessary, it is not supported. If you need to customize any element, consider the following issues:

- 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. If you made any code changes to the elements in the asset type directory, they are overwritten when AssetMaker moves the new copies.
- The upgrade from one version of CS-Direct to another typically installs new source elements in the `AssetStubElementCatalog` table. That means that any code changes in the stub elements are overwritten when you re-register your asset types after you upgrade.

Be scrupulous about tracking all of your customizations at all times. That way you can re-create your work if you need to.

About PreUpdate and PostUpdate

The actions or procedures that can be performed on any asset type are called **functions**. For example, to create, edit, copy, delete, and so on are all functions.

The `PreUpdate` and `PostUpdate` elements contain logic that is invoked before and after writing information about an asset to the database. The `PreUpdate` and `PostUpdate` elements uses a variable named `updatetype`. to determine the kind of function that is underway. If necessary, you can customize these elements, using the value of `updatetype` as a condition for additional logic.

The functions that invoke the `PreUpdate` and `PostUpdate` elements are as follows:

- new
- edit
- delete
- XMLPost

Both the new function and the edit function call the `PreUpdate` element twice:

- Before the function renders the "New" or "Edit" form
- Before it saves an asset

The following table defines the values of the `updatetype` variable:

Table 3:

updatetype value	Description
`setformdefaults`	For `PreUpdate` only: the new function is about to render the "New" form. (Note that `updatetype` is never set to `setformdefaults` in the `PostUpdate` element.)
`create`	For `PreUpdate`: the new function is about to save a new asset. For `PostUpdate`: the new function saved a new asset.

Table 3:

updatetype value	Description
`editfront`	For `PreUpdate` only: the edit function is about to render the "Edit" form. (Note that updatetype is never set to `editfront` in the `PostUpdate` element.)
`edit`	For `PreUpdate`: the edit function is about to save the edited asset. For `PostUpdate`: the edit function just saved an edited asset.
`delete`	For `PreUpdate`: the delete function is about to delete an asset. For `PostUpdate`: the delete function just deleted an asset.
`remotepost`	For `PreUpdate`: the XMLPost function is about to import an asset. For `PostUpdate`: the XMLPost function just imported an edited asset.
`MSWord`	For `PreUpdate`: CS-Desktop is about to save an asset created or edited in Microsoft Word.

There are several reasons why you might need to modify the `PreUpdate` or `PostUpdate` elements. For example, perhaps your system is set up to import batches of articles from a wire service. You can modify the `PreUpdate` element to set the value for **Source** to "wirefeed" and the value for a **Byline** field to "API" when `updatetype=remotepost`.

Step 9: (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 icon.
5. Click **Delete Subtype.**

Step 10: (Optional) Configure Association Fields

Named associations are described in "The Basic Asset Model" on page 190. 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.

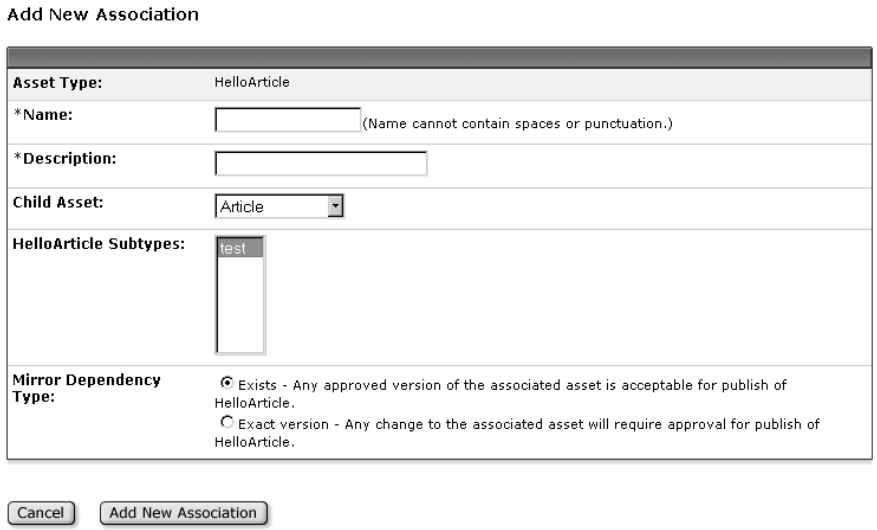
Examples of association fields from the Burlington Financial sample site include the Main Image and Teaser Image associations between article assets and imagefile assets. When you create a Burlington Financial article asset, you can select an image asset from the **Main Image** and **Teaser Image** fields.

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 513).
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 6.

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 11: (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.

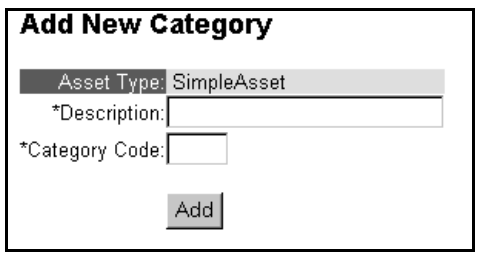
Although all basic asset types have a **Category** field (column) by default, it is not a required field and you do not have to use it.

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 12: (Optional) Configure Sources

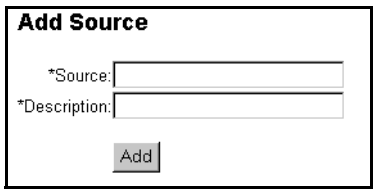
Sources apply to all the asset types in all the sites on your system. Therefore, if you are using Source, you need to add your sources only once (not for each asset type).

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 13: (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 14: (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 195). 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`
- `OpenMarketXcelerate/AssetType/`*`YourAssetType`*`/IndexCreateVerity.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 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 23, "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), you migrate 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 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 15

Designing Flex Asset Types

As mentioned in Chapter 10, “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 211.) 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 211, 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 16, “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 16, “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 208. 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 280. 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 or new member of the flex family.
2. Configure the development system so that you have easy access to the new asset types:
   a. Enable the new asset types on all the Content Server sites on the development system.
   b. Create **Start Menu** shortcuts for all the new asset types.
   c. Put the new flex definition, flex parent definition, and attribute types on the **Design** tab.
   d. Create a tab for your new flex parent and flex asset types.
3. Create the flex attributes and design your attribute editors. For information about attribute editors, see Chapter 16, "Designing Attribute Editors."
4. Create the flex filter assets.
5. Create the flex parent definitions.
6. Create the flex definitions.
7. Create the flex parents.
8. Test your design by creating enough flex assets to examine the data structure that you have designed. (Procedures for creating assets are presented in the *Content Server User's Guide*.)
9. Create templates for the flex assets, the flex member of the flex family. This step is described in Chapter 20, "Creating Template, CSElement, and SiteEntry Assets" and Chapter 23, "Coding Elements for Templates and CSElements."
10. Move your asset types to other systems (management and delivery). This step is described in the *Content Server Administrator's Guide*.

Before You Begin

Be sure to 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*.

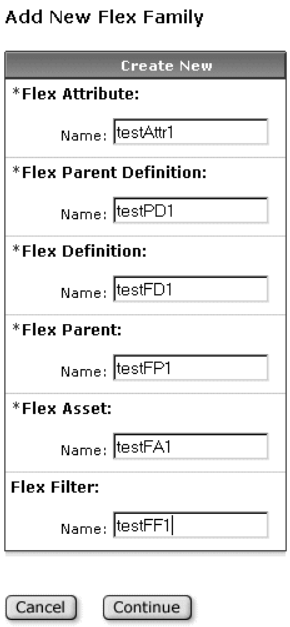
Step 1: Create a Flex Family or a New Flex Family Member

To create a new flex family or family member

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 Flex Family Maker form appears:



4. For **Flex Attribute**, do one of the following:
   - If you are creating a completely new flex family — all five members — click in the **Name** field and enter the name of the new flex attribute type.

     The name you enter in this field is the internal name of the new attribute type. It becomes the name of the core table for this asset type and the prefix for all its auxiliary tables.
   - If you are creating a family that shares attributes with another family, select the appropriate attribute type from the drop-down field.
5. For **Flex Parent Definition,** do one of the following:
   - If you are creating a completely new flex family — all five members — or creating a new family that shares attributes with another family, click in the **Name** field and enter the name of the new flex parent definition.

     The name you enter in this field is the internal name of the new parent definition type. It becomes the name of the core table for this asset type and the prefix for all its auxiliary tables.
   - If you are creating a family that shares attributes and parents, select the appropriate parent definition from the drop-down field.
6. For **Flex Definition**, click in the **Name** field and enter the name of the new flex definition.

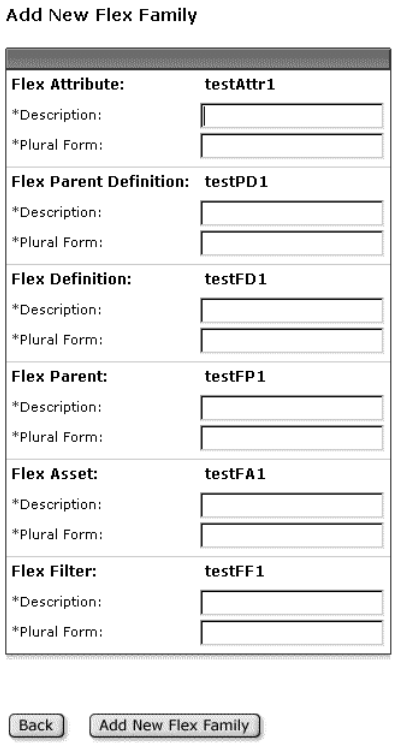
   The name you enter in this field is the internal name of the new flex definition type. It becomes the name of the core table for this asset type and the prefix for all its auxiliary tables.
7. For **Flex Parent**, do one of the following:
   - If you are creating a completely new flex family — all five members — or creating a new family that shares attributes with another family, 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 type. It becomes the name of the core table for this asset type and the prefix for all its auxiliary tables.
   - If you are creating a family that shares attributes and parents, select the appropriate parent from the drop-down field.
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 208.

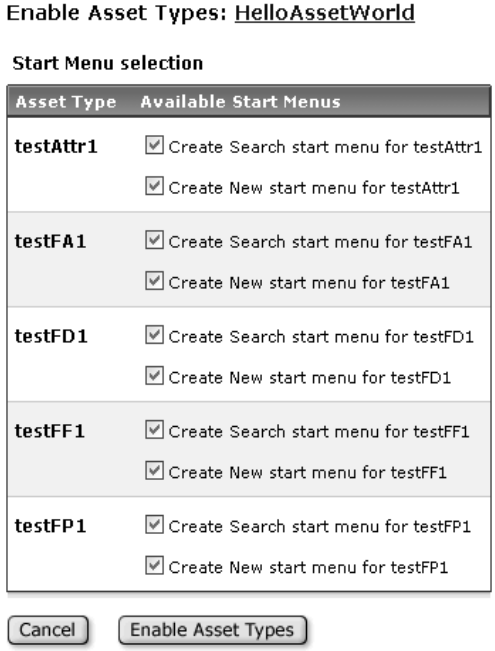
    It also copies elements that format the forms for assets of these types to a directory with the name of the asset type in the `ElementCatalog` and `SystemSQL` tables.

Step 2: 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 3: 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`

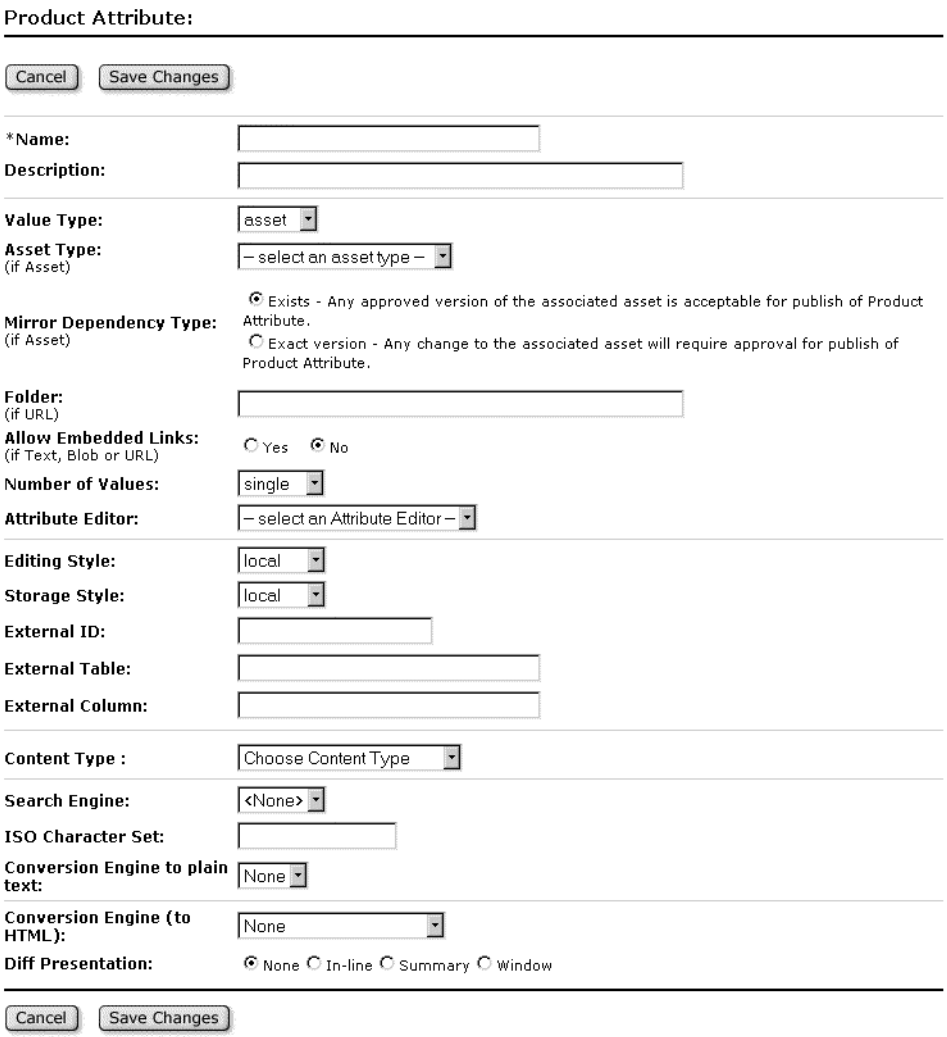
> **Note**
>
> The `url` data type has been deprecated in the 4.0 version of the product. Use the `blob` data type instead.

- Creating attributes of type `asset`
- Creating foreign attributes (that is, attributes that are stored in a foreign table)

Creating Flex Attributes: the 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 **New 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 **asset** or **blob**, see "Creating Flex Attributes of Type Asset" on page 324 or "Creating Flex Attributes of Type Blob (Upload Field)" on page 323, as appropriate.)

   If you need help deciding which data type is appropriate for your attribute, see "Data Types for Attributes" on page 202.

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 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.

   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).

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 203.

   For information about creating attribute editors, see Chapter 16, "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 345.

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: the Basic Procedure" on page 322.

2. Click in the **Value Type** field and select `blob`.

3. (Optional) Click in **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: the Basic Procedure" on page 322.

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 345.
6. Complete steps 8 through 10 of the procedure "Creating Flex Attributes: the Basic Procedure" on page 322.

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: the Basic Procedure" on page 322.
2. Click in the **Value Type** field and select `asset`.
3. Click in the **Asset Type** field and select one 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 311 for information about appropriate attribute editors.
7. Complete steps 8 through 10 of the procedure "Creating Flex Attributes: the Basic Procedure" on page 322.

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.

Before you begin, be sure to register the foreign table with Content Server. For information, see "Registering a Foreign Table" on page 231.

To create a foreign attribute

1. Complete steps 1 through 6 in the procedure "Creating Flex Attributes: the Basic Procedure" on page 322. Note that you **cannot** select either **asset** or **blob** (or url) in the **Value Type** field.
2. (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 Type** field, click in the **Attribute Editor** field and select an appropriate one.

3. 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**.
4. Click in the **Storage Style** field and select **external** from the drop-down list.
5. 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.
6. Click in the **External Table** field and enter the name of the table that stores this attribute.
7. 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.
8. Complete steps 9 through 11 of the procedure "Creating Flex Attributes: the Basic Procedure" on page 322.

Step 4: (optional) Create Flex Filter Assets

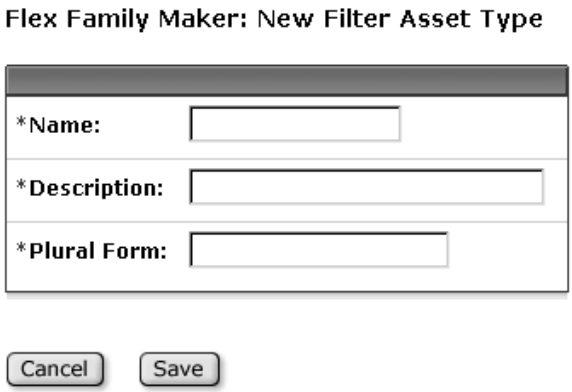
Before you can create flex filter assets, the flex attributes that you plan to use as the input and output attributes must already be created. If the appropriate flex attributes do not exist yet, create them before continuing with this procedure. Note the following requirements:

- 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 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. Click **New** and then select the name of your flex definition asset type from the list of shortcuts.

The **New** filter form appears:



3. In the **Name** field, enter a name of for the flex filter asset. You can use up to 64 characters.
4. In the **Description** field, enter a description of the filter asset.
5. In the **Filter** field, select one of the registered filters from the drop-down list. By default, there is only one registered filter: **Document Transformation**. For information about this filter type, see “Flex Filters” on page 206.

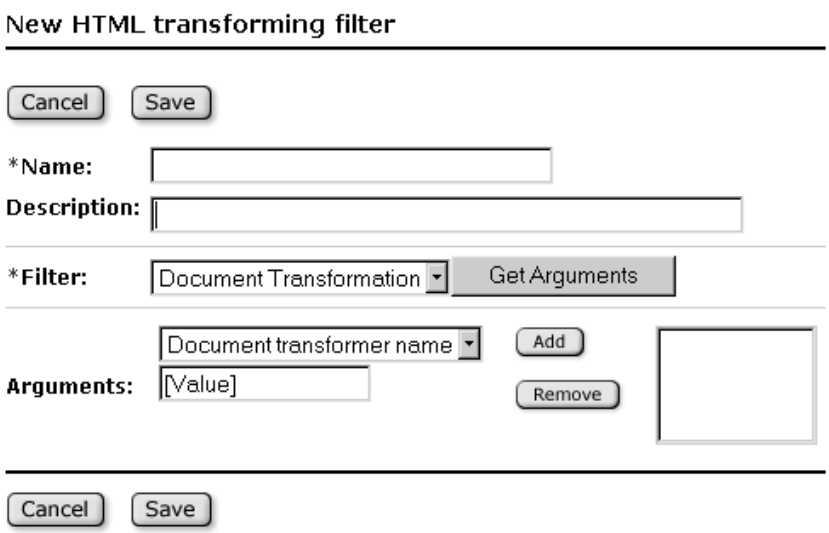
Note

If your system has any custom filters registered, they will also appear in this drop-down list. Custom filters use custom arguments.

This procedure describes how to configure values for the Document Transformation filter. If you select a custom filter, be sure that you specify appropriate values for its arguments.

6. Click **Get Arguments**.

   The **New Filter** form displays the **Arguments** fields:



7. Specify the **Document transformer name** argument for the **Document Transformation** filter as follows:
   a. From the top **Arguments** field, select **Document transformer name** from the drop-down list.
   b. Click in the **Arguments** ([Value]) field and enter the following text, exactly as it is presented:

      ```
      Verity: Convert to HTML
      ```

   c. Click **Add**.

> **Note**
>
> By default, the **Verity: Convert to HTML** transformation engine is the only registered engine that you can specify for the **Document transformer name** argument. For information about configuring the Verity transformation engine so that it can also convert documents to XML, see “Registering a New Transformation Engine” on page 338.

8. Specify the **Output document extension** argument for the **Document Transformation** filter as follows:
   a. From the top **Arguments** field, select **Output document extension** from the drop-down list.
   b. Click in the **Arguments** ([Value]) field and enter: `.htm`
   c. Click **Add**.

9. Specify the **Input attribute name** argument for the **Document Transformation** filter as follows:
   a. From the top **Arguments** field, select **Input attribute name** from the drop-down list.
   b. Click in the **Arguments** ([Value]) field and enter the name of the flex attribute whose contents will be converted to HTML by this filter and then stored in the output variable. This attribute must already exist and it must be of type `blob`.
   c. Click **Add**.
10. Specify the **Output attribute name** argument for the **Document Transformation** filter as follows:
    a. From the top **Arguments** field, select **Output attribute name** from the drop-down list.
    b. Click in the **Arguments** ([Value]) field and enter the name of the flex attribute that holds the data that the filter converts to HTML. This attribute must already exist and it must be of type `blob`.
    c. Click **Add**.
11. Click **Save**.

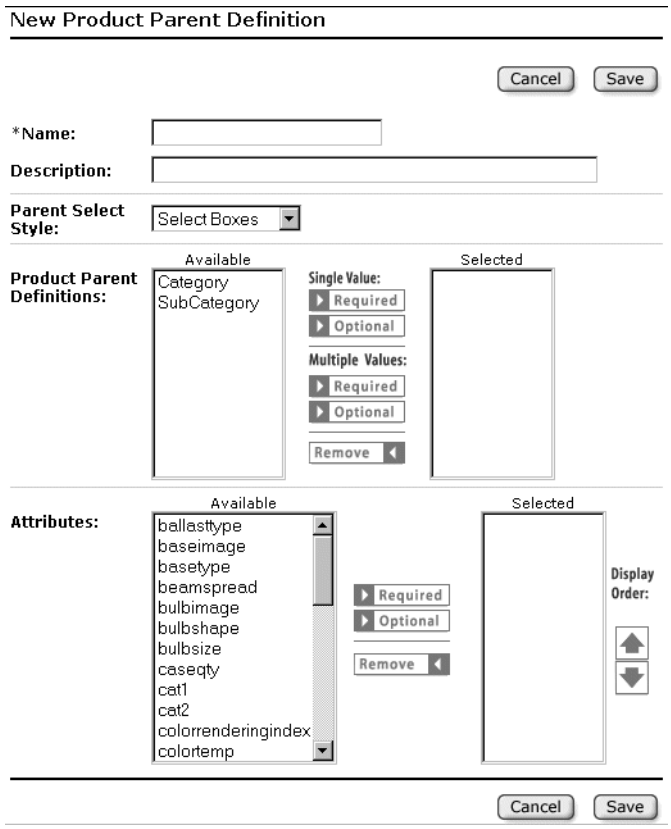
    This filter will now appear in the **Filter** list on the **New** and **Edit** forms for your flex parent definition and flex definition assets.

Step 5: 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 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 313.

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.

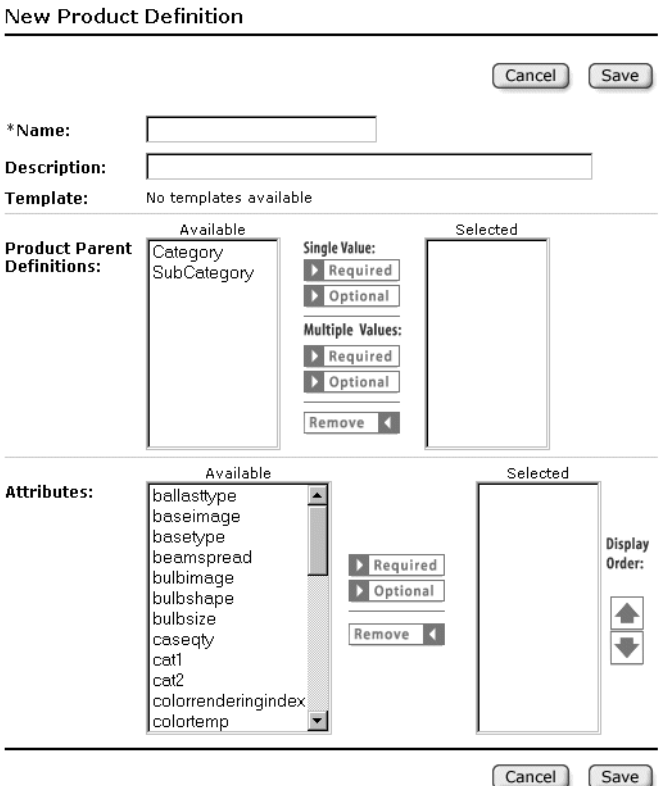
8. Repeat steps 6 and 7 as many times as necessary. Remember that the corresponding **New** parent form will include a field for each item that you select in the **Available** list on this parent definition form.
9. In the **Attributes** section, select the appropriate attributes. Note that if you are 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, that 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 5: 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 313.
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.

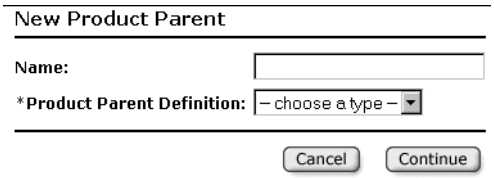
7. Repeat steps 5 and 6 as many times as is necessary. Remember that the corresponding **New** flex asset form will include a field for each item that you select in the **Available** list on this flex definition form.
8. In the **Attributes** section, select an attribute. Note that if you are going to 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 6: Create Flex Parent Assets

To create flex parent assets

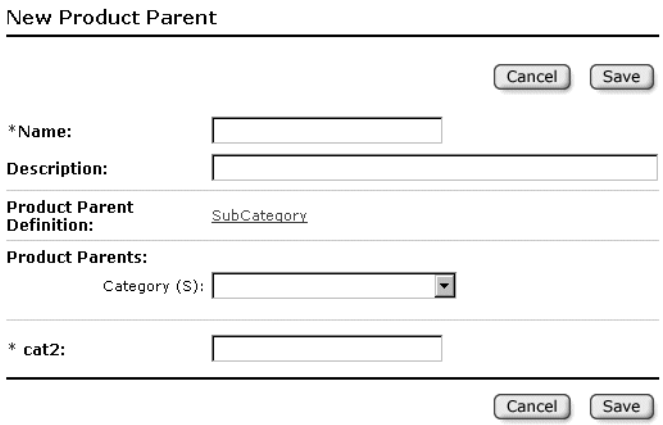
1. If the Content Server interface is not open, log in and select the appropriate site.
2. Click **New** and then select the name of your flex parent asset type from the list of shortcuts.

   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 depends 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 8: 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, "Coding Elements for Templates and CSElements."

Step 9: Test Your Design (Create Test Flex Assets)

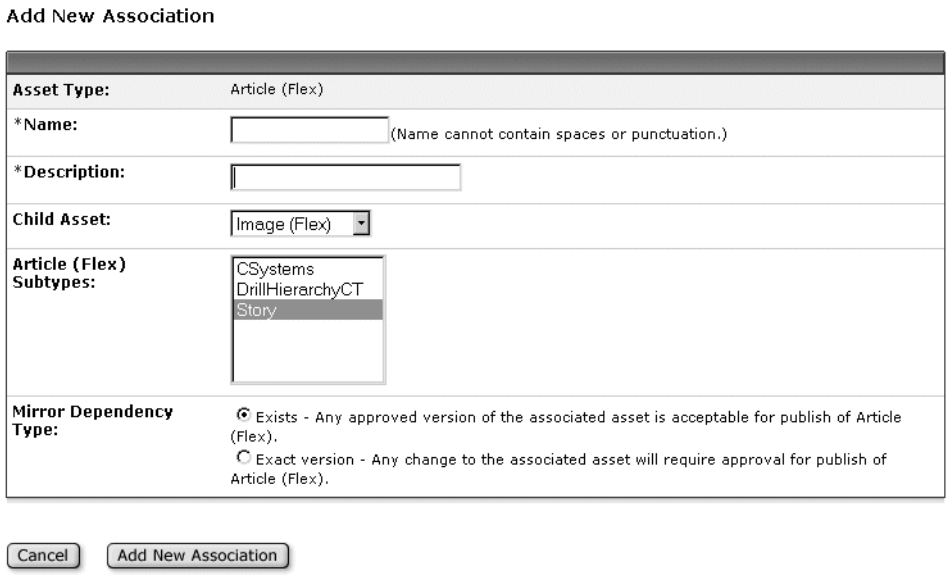
To thoroughly test your design, you must create some flex assets so that you can examine where they appear on the tree, what their forms look like, how long it takes to load their forms, and so on.

For information about creating new assets, see the *Content Server User's Guide*.

Step 10 (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 11: 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.

Custom Filter Classes or Transformation Engines

This section describes how to register a new filter class or transformation engine.

Registering a New Filter Class

To register a new filter class

1. Copy your `.jar` or class file to the directory that holds the Content Server product jars:
   - 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, as follows:
   a. Select the `Filters` table.
   b. Either select **File** > **New** > **Record**, or select **New** from the right-mouse menu.
   c. Enter the name of the filter class in the `name` column.
   d. Enter a description of the filter class in the `description` column.
   e. Enter the exact name of the filter class in the `classname` column.
   f. Select **File** > **Save**.

   Your filter class is registered and will now be displayed in the **Filter** drop-down list in the **New** and **Edit** forms for filter assets.

Registering a New Transformation Engine

The Document Transformation filter class can invoke any registered transformation engine, that is, a transformation engine with an entry in the `SystemTransforms` table. By default, Verity Keyview is the only transformation engine that is installed. Also by default, this engine is configured to convert documents to HTML.

If appropriate, you can register your own transformation engines to use with the Document Transformation filter class. For example, because the Verity Keyview engine is also capable of converting documents to XML, you can create another entry for the Verity Keyview engine and configure it to convert documents to XML.

To register a new transformation engine

> **Note**
>
> If you are adding another instance of the Verity Keyview engine to the `SystemTransforms` table so that you can configure it to convert documents to XML, do not complete step 1of this procedure. The Keyview class file is already in the appropriate place so you should start with step 2.

1. Copy the .jar or class file of the transformation engine to the directory that holds the Content Server product jars:
   - 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 `SystemTransforms` table for the new transformation engine, as follows:
   a. Select the `SystemTransforms` table.
   b. Either select **File** > **New** > **Record**, or select **New** from the right-mouse menu.
   c. In the `name` column, enter the name of the transformation engine. For example: `Verity_Convert_to_XML`
   d. In the `description` column, enter a description of the engine. For example: `Convert to XML using Verity Keyview`
   e. In the `target` column, enter `text/`*`filetype`*. For example: `text/XML`
   f. In the `classname` column, enter the **exact** name of the engine class. For example, if you are creating an additional entry for the Verity Keyview engine, copy and paste the `classname` from the `Verity: Convert to HTML` row. (Its classname is `com.openmarket.Transform.Keyview.KeyviewTransform`.)
   g. In the `args` column, set any arguments that are appropriate for this transformation engine. For example, if you are creating an additional entry for the Verity Keyview engine, enter `exporttype=XML`
   h. Select **File** > **Save**.

   Your transformation engine is registered. You can now use this transformation engine with your Document Transformation filter assets.

Chapter 16

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 203. 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 351).

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 351. 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 Products 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 348.

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 Products 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 346.

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 Products 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 348.

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`.

The following attribute editor code defines the `XSIZE` as 40 pixels, the `YSIZE` as 5 pixels, and disables text wrapping by setting `WRAPSTYLE` to `OFF`:

```
<?XML VERSION="1.0"?>
<!DOCTYPE PRESENTATIONOBJECT SYSTEM "presentationobject.dtd">

<PRESENTATIONOBJECT NAME="TextAreaTest">
   <TEXTAREA XSIZE="40" YSIZE="5" WRAPSTYLE="OFF">
   </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 HTML editor, a third-party product available from Ektron, Inc. 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: • When the tree is displayed, the attribute uses the "pick from tree" mechanism. • 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 203.) 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 357.

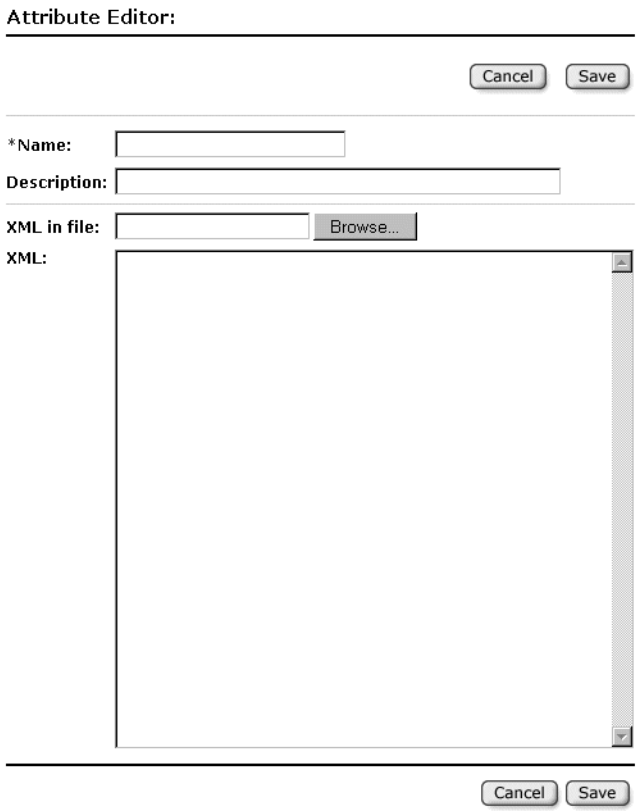
Creating Attribute Editors

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 345 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 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 343.

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 351.

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 -->
```

```
<IF COND="Variables.AttrType=url">
<THEN>
<setvar NAME="errno" VALUE="0"/>
<BEGINS STR="AttrValueList.urlvalue"
WHAT="AttrValueList."/>
<IF COND="Variables.errno=1">
<THEN>
<setvar NAME="filename" VALUE="CS.UniqueID.txt"/>
</THEN>
<ELSE>
<setvar NAME="filename"
VALUE="AttrValueList.urlvalue"/>
</ELSE>
</IF>

<INPUT TYPE="hidden" NAME="Variables.AttrName_file"
VALUE="Variables.filename"
REPLACEALL="Variables.AttrName,Variables.filename"/>

<setvar NAME="errno" VALUE="0"/>
<BEGINS STR="AttrValueList.@urlvalue"
WHAT="AttrValueList."/>
<IF COND="Variables.errno=1">
<THEN>
<setvar NAME="MyAttrVal" VALUE="Variables.empty"/>
</THEN>
<ELSE>
<setvar NAME="MyAttrVal"
VALUE="AttrValueList.@urlvalue"/>
</ELSE>
</IF>
</THEN>
</IF>

<!-- Display a TEXTAREA with all parameters from Attribute
--Editor xml -->
<!-- The NAME of the input must be the attribute name -->
<IF COND="Variables.disableTextArea=yes">
<THEN>
<TEXTAREA DISABLED="yes" NAME="Variables.AttrName"
ROWS="Variables.YSIZE" COLS="Variables.XSIZE"
WRAP="Variables.WRAPSTYLE"
REPLACEALL="Variables.AttrName,Variables.XSIZE,
Variables.YSIZE,Variables.WRAPSTYLE,Variables.empty">

<!-- For most single valued attrs, the value is contained
   in
MyAttrVal -->
<csvar NAME="Variables.MyAttrVal"/>
</TEXTAREA>
</THEN>
<ELSE>
```

```
<TEXTAREA NAME="Variables.AttrName"
ROWS="Variables.YSIZE" COLS="Variables.XSIZE"
WRAP="Variables.WRAPSTYLE"
REPLACEALL="Variables.AttrName,Variables.XSIZE,
Variables.YSIZE,Variables.WRAPSTYLE,Variables.empty">
<!-- For most single valued attrs, the value is
contained in MyAttrVal -->
<csvar NAME="Variables.MyAttrVal"/>
</TEXTAREA>
</ELSE>
</IF>
</then>
<else>
<!-- Multiple valued attributes -->
<!-- For single value attributes we can usually use the
default RequireInfo -->
<!-- For multiple value attributes we need to append to
RequireInfo for each value -->
<if COND="Variables.RequiredAttr=true">
<then>
<setvar NAME="RequireInfo"
VALUE="Variables.RequireInfo*Counters.TCounterVariables.
AttrName*ReqTrue*Variables.AttrType!"/>
</then>
<else>
<setvar NAME="RequireInfo"
VALUE="Variables.RequireInfo*Counters.TCounterVariables
.AttrName*ReqFalse*Variables.AttrType!"/>
</else>
</if>

<!-- Display a TEXTAREA with all parameters from Attribute
Editor xml -->
<!-- The NAME of the input must be the attribute name
prepended by the TCounter counter -->
<IF COND="Variables.disableTextArea=yes">
<THEN>
<TEXTAREA DISABLED ="yes"
   NAME="Counters.TCounterVariables.AttrName"
ROWS="Variables.YSIZE" COLS="Variables.XSIZE"
WRAP="Variables.WRAPSTYLE"
REPLACEALL="Counters.TCounter,
Variables.AttrName,Variables.XSIZE,
Variables.YSIZE,Variables.WRAPSTYLE">
<csvar NAME="Variables.tempval"/> </TEXTAREA>
</THEN>
<ELSE>
<TEXTAREA NAME="Counters.TCounterVariables.AttrName"
ROWS="Variables.YSIZE" COLS="Variables.XSIZE"
WRAP="Variables.WRAPSTYLE"
REPLACEALL="Counters.TCounter,
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 17

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 inport 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 18, “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 18, "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.

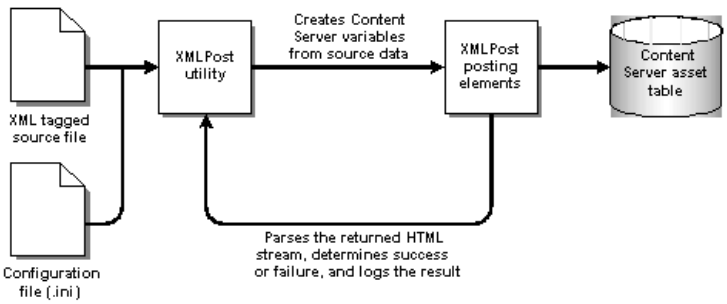
   This file also sets several configuration properties, including the name of the SiteCatalog entry for the posting element that you want XMLPost to use. For all assets, the name of this posting element is RemoteContentPost. For information about the posting elements for flex assets, see Chapter 18, "Importing 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. The file extension for your source files. Typically set to `xml`. For example: `xmlpost.xmlfilenamefilter: .xml`
`xmlpost.proxyhost`	Optional. 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. For example: `xmlpost.proxyhost: nameOfServer`

Property	Description
`xmlpost.proxyport`	Optional. 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. For example: `xmlpost.proxyport: 80`

`xmlpost.url`	Required.
	The first part of the URL for the page entry of the posting element.
	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).
	The value that you set for this property should use the following convention:
	• The name of the server that holds the Content Server database.
	• 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/`.
	• The name of the ContentServer servlet.
	For example:
	`xmlpost.url: http://servername/servlet/ContentServer`
	or
	`xmlpost.url: http://servername/NASapp/cs/ContentServer`

`xmlpost.logfile`	Optional. The name of the file to log the results of importing (posting) each source file. 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. For example: `xmlpost.logfile: ArticlePost.txt`

Property	Description
`xmlpost.success`	Optional. The string to look for in the response to determine if the post was a success. For example: `xmlpost.success: Success!`
`xmlpost.failure`	Optional. The string to look for in the response to determine if the post was a failure. For example: `xmlpost.failure: Error`
`xmlpost.deletefile`	Optional. 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. For example: `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. 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. For example: `xmlpost.numargs: 7` Note that you can also specify your own custom variables with these name/value pairs.

`xmlpost.argname1: pagename`	Required.
	The pagename for the `RemoteContentPost` element. Typically the `pagename` argument is specified as `xmlpost.argname1`.
	For example:
	`xmlpost.argname1: pagename`
	`xmlpost.argvalue1: OpenMarket/Xcelerate/Actions/RemoteContentPost`

`xmlpost.argname2: AssetType`	Required.
	The asset type of the assets that are defined in the source files. Typically, `AssetType` is specified as `xmlpost.argname2`.
	For example:
	`xmlpost.argname2: AssetType`
	`xmlpost.argvalue2: Collection`
	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.
	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`.
	For example:
	`xmlpost.argname3: authusername`
	`xmlpost.argvalue3: editor`
	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.
	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`.
	For example:
	`xmlpost.argname4: authpassword`
	`xmlpost.argvalue4: xceleditor`

`xmlpost.argname5: xmlpostdebug`	Optional
	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.
	You can set this property to any value. For example:
	`xmlpost.argname5: xmlpostdebug`
	`xmlpost.argvalue5: on`
	Note: Be sure to include a value for the `xmlpost.logfile` property if you enable debugging.

`xmlpost.argname6: inifile`	Optional.
	The name of the `ini` file to use when connecting to the Content Server database. Typically, `inifile` is specified as `xmlpost.argname5`.
	For example:
	`xmlpost.argname6: inifile`
	`xmlpost.argvalue6: futuretense.ini`

Property	Description

`xmlpost.argname7: publication`	Optional.
	Athough using this property is optional, you must specify a site for each asset that you are importing.
	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.
	For example:
	`xmlpost.argname7: publication`
	`xmlpost.argvalue7: Burlington Financial`
	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 371 for more information.

`xmlpost.argname8: startmenu`	Optional.
	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.)
	For example:
	`xmlpost.argname8: startmenu`
	`xmlpost.argvalue8: New Article`
	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 371 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` (yes or no)	Optional. Specifies that a source file will provide a site name that identifies which site the asset belongs to. For example: postpublication: y 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` (yes or no)	Optional. 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. For example: postprimarypubid: y
`postpublist`	`y` or `n` (yes or no)	Optional. Specifies that a source file will provide a list of sites that the asset is shared with. For example: postpublist: y
`poststartmenu`	`y` or `n` (yes or no)	Optional. Specifies that a source file will provide a value for the Start Menu short cut that places the asset into a workflow process. For example: 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` (yes or no)	Required. Specifies the name of the tag. The name should exactly match the name of the field that it represents. For example, the tag property for a **name** field is: `postname: y`
`trunc`*`tagname`*	*`N`* (integer)	Optional. Whether to truncate the data in the source file marked by this tag. For example: `truncname: 64` 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` (yes or no)	Optional. 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). For example: `notrimname: y` If you do not specify this property, XMLPost trims the white space for the tag by default.

Tag property	Value	Description
multi*tagname*	combine or *separate*	Required if the same tag is used more than once in a single source file. 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). 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. For example, for a keyword field (column): • If you set multikeyword: combine, XMLPost stores all the values marked by a keyword tag to the same keyword variable. • 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. 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.* For example: multikeyword: separate seedkeyword: 1

Tag property	Value	Description

file*tagname*	y or n
	(yes or no)
		Required if the tag represents an upload field (a URL column or BLOB).
		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.
		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:
		`posturlpicture: y`
		`fileurlpicture: y`
		Then, in the source file for an imagefile asset, you specify the value for the `urlpicture` field like this:
		`<urlpicture>`*`relative_path_to/filename.jpg`*
		`</urlpicture>`
		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 375:

```
<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 in to.)
- 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 381 for more information.

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 -cp installdir\cs.jar;installdir\MSXML.jar;
      installdir\commons-logging.jar;
      installdir\cs-core.jar;installdir\commons-codec-
      1.3.jar;installdir\commons-httpclient-3.0-
      rc2.jar;installdir\commons-lang-2.1.jar
      COM.FutureTense.XML.Post.XMLPostMain
         -sSourcefile.xml -cConfigfile.ini
   ```

   Solaris

   ```
   % java -cp installdir/cs.jar:installdir/MSXML.jar:
      installdir/commons-logging.jar:
      installdir/cs-core.jar: installdir\commons-codec-
      1.3.jar:installdir\commons-httpclient-3.0-
   ```

```
rc2.jar:installdir\commons-lang-2.1.jar
COM.FutureTense.XML.Post.XMLPostMain
-sSourcefile.xml -cConfigfile.ini
```

where *installdir* is your FatWire installation directory. (Note that there are several options for designating the source file. See "Options for Identifying Source Files" on page 379 for information.)

Note

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 -cp installdir/cs.jar:installdir/MSXML.jar:
   installdir/commons-logging.jar:
   installdir/cs-core.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 -cp installdir/cs.jar:installdir/MSXML.jar:
   installdir/commons-logging.jar:
   installdir/cs-core.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 a `.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 -cp installdir\cs.jar;installdir\MSXML.jar;
   installdir\commons-logging.jar;
   installdir\cs-core.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 (which requires you to run the utility separately for each asset type because you must identify a different configuration file for each), it is convenient to run XMLPost from a batch file.

In the batch file, include a command line statement for each asset type: a statement that 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:

- The server name specified in the xmlpost.URL property setting in your configuration file is not a valid server name. Examine this property and make sure that the server name is correct.
- 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 this error code is when the data that XMLPost tries to save to a specific column (field) is too large for that column. Resolving this depends on your goals. If it is okay 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 7, "Content Server Tools and Utilities."

Chapter 18

Importing Flex Assets

Chapter 17, “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 19, “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 19, "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 378.

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 389.
2. Flex attributes. The configuration file must instruct XMLPost to call the `RemoteContentPost` element. For information, see "Flex Attributes" on page 391.
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 395.

   Flex definitions. The configuration file must instruct XMLPost to call the `RemoteContentPost` element. For information, see "Flex Definitions and Flex Parent Definitions" on page 395.

   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 399.
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 19, "Importing Flex Assets with the BulkLoader Utility."
   - Use XMLPost. See "Importing Flex Assets with XMLPost" on page 400.

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.

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 17, "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 399 and "Importing Flex Assets with XMLPost" on page 400 for information about which posting element to use.)

In both cases, you create configuration files and source files (as described in "Overview" on page 386 and supplemented in this chapter), and then invoke the XMLPost utility (as described in "Using the XMLPost Utility" on page 378).

Note

For reference, sample XMLPost 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.

Internal Names vs. External Names

When you create your flex family of asset types (see "Step 1: Create a Flex Family or a New Flex Family Member" on page 317), 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 17, "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:
`<name>`
property:
`postname`
	Required.
	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:
`<description>`
property:
`postdescription`
	Optional.
	Description of the use or function of the attribute.

Attribute editor tag and property	Description
tag: `<AttrTypeText>` property: `postAttrTypeText`	Required. Either the name of the file with the attribute editor XML code, or the actual code. 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:
`<name>`
property:
`postname`
	Required.
	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:
`<description>`
property:
`postdescription`
	Optional.
	Description of the use or function of the attribute.

tag:
`<valuestyle>`
property:
`postvaluestyle`
	Optional.
	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.
	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`.

Flex attribute tag and property	Description

tag:
`<type>`
property:
`posttype`
	Required.
	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 202.

tag: `<assettypename>` property: `postassettypename`	Required if `<type>` is set to `asset`. The name of the asset type that the attribute holds.

tag:
`<upload>`
property:
`postupload`
	Required if `<type>` is set to `blob` (or `url`, which is deprecated in 4.0).
	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>` property: `postattributetype`	Optional. The name of the attribute editor to use, if applicable.

tag:
`<enginename>`
property:
`postenginename`
	Optional.
	If you are using a search engine on your management system, the name of the search engine.

tag:
`<charsetname>`
property:
`postcharsetname`
	Optional.
	The search engine character set to use. By default, it is set to ISO 8859-1.

tag:
`<editing>`
property:
`postediting`
	Foreign attributes only.
	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:
`<storage>`
property:
`poststorage`
	Foreign attributes only.
	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:
`<externalid>`
property:
`postexternalid`
	Foreign attributes only.
	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.

Flex attribute tag and property	Description
tag: <externalcolumn> property: postexternalcolumn	Foreign attributes only. The name of the column in the foreign table that holds the values for this attribute.
tag: <externaltable> property: postexternaltable	Foreign attributes only. The name of the foreign table that contains the columns identified by externalid and external column.
tag: <publication> property: postpublication	Optional. 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 3: Create Flex Attributes” on page 321.

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:
`<internalname>`
property:
`postinternalname`
	Required.
	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:
`<internaldescription>`
property:
`postinternaldescription`
	Optional.
	The description of the use or function of the asset.

tag:
`<renderid>`
property:
`postrenderid`
	Optional. For flex definitions only.
	The ID of the Template asset that is to be assigned to all the flex assets that are created with this flex definition.

tag:
`<parentselectstyle>`
property:
`postparentselectstyle`
	Optional. For flex parent definitions only.
	Defines how flex parents are to be selected when a user creates a flex asset using the definition.
	This property/tag represents the **Parent Select Style** field in the **New** and **Edit** parent definition forms.
	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 4: (optional) Create Flex Filter Assets” on page 325 and “Step 5: Create Flex Definition Assets” on page 330.	

tag:
`<OptionalSingleParentList>`
property:
`postOptionalSingleParentList`
	Use this tag to specify any single optional parent definition.

Flex definition and flex parent definition tag and property	Description

tag:
`<RequiredSingleParentList>`
property:
`postRequiredSingleParentList`
	Use this tag to specify any single required parent definition.

tag:
`<RequiredMultipleParentList>`
property:
`postRequiredMultipleParentList`
	Use this tag to specify more than one required parent definitions.

tag:
`<OptionalMultipleParentList>`
property:
`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 4: (optional) Create Flex Filter Assets” on page 325 and “Step 5: Create Flex Definition Assets” on page 330.	

tag:
`<RequiredAttrList>`
property:
`postRequiredAttrList`
	The list of attributes that are required for the flex parents or the flex assets that use the definition.

tag:
`<OptionalAttrList>`
property:
`postOptionalAttrList`
	The list of attributes that are optional for the flex parents or the flex assets that use the definition.

tag:
`<OrderedAttrList>`
property:
`postOrderedAttrList`
	The order in which all attributes, be they required or optional, should appear in the “New,” “Edit,” “Inspect,” and similar forms.
	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 400.

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:
`<_ASSET_>`
property:
`post_ASSET_`
	Required.
	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:
`<_TYPE_>`
property:
`post_TYPE_`
	Required.
	The name of the flex definition that this flex asset is using.

tag:
`<_ITEMNAME_>`
property:
`post_ITEMNAME_`
	Required.
	The name of the asset.

tag:
`<_ITEMDESCRIPTION_>`
property:
`post_ITEMDESCRIPTION_`
	Optional.
	The description of the asset.

Tag and property	Description
tag: <_GROUP_parentDefinitionName> property: post_GROUP_parentDefinitionName	Optional. 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: post_Group_Department post_Group_Division See "Specifying the Parents of a Flex Asset" on page 408 for more information about using this tag and property.
tag: <_GROUPDESCRIPTIONS_> property: post_GROUPDESCRIPTIONS_	Optional. If the parent that you are designating is new, you can also include the description of the parent definition.
tag: <displaytype> property: postdisplaytype	Optional. The name of the Template asset for the flex asset.
tag: <AttributeName> property: 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. 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: <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:
`<_DEFINITION_>`
property:
`post_DEFINITION_`
	Required.
	The name of the flex definition that this flex asset is using.
	(Note that `post_TYPE` will also work.)

tag:
`<_ITEMNAME_>`
property:
`post_ITEMNAME_`
	Required.
	The name of the asset.

tag:
`<_ITEMDESCRIPTION_>`
property:
`post_ITEMDESCRIPTION_`
	Optional.
	The description of the asset.

`tag:`
`<ParentList>`
property:
`post_ParentList`
	Optional.
	The flex asset's immediate parents.

tag:
`<template>`
property:
`posttemplate`
	Optional.
	The name of the Template asset for the flex asset.
	(Note that `postdisplaytype` will also work.)

tag:
`<AttributeName>`
property:
`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.
	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:
	`<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 406.

```
<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 but does not specify a value for it.

For example:

```
<bulbsize></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=: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=</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 19

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:

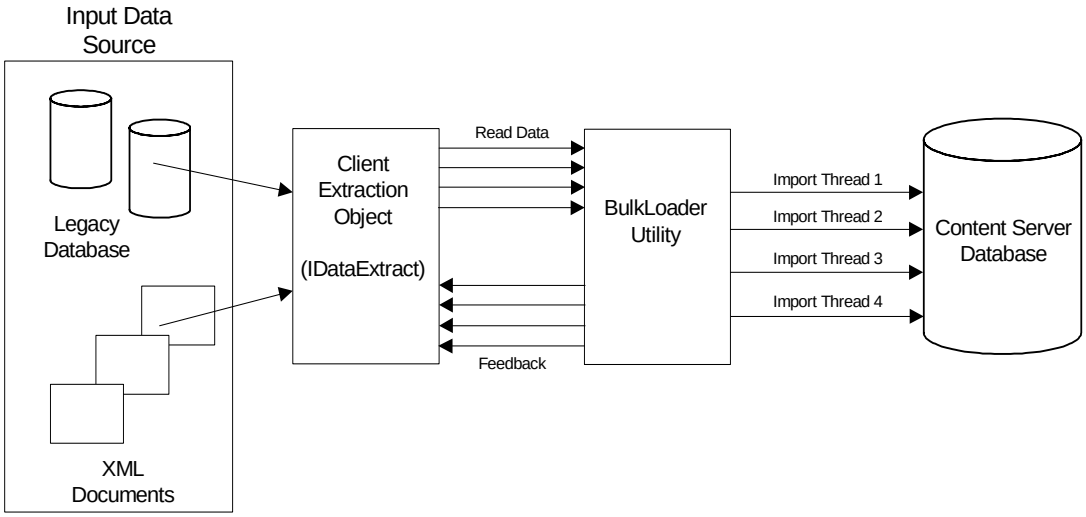


Figure 6: Client-specific implementation of BulkLoader

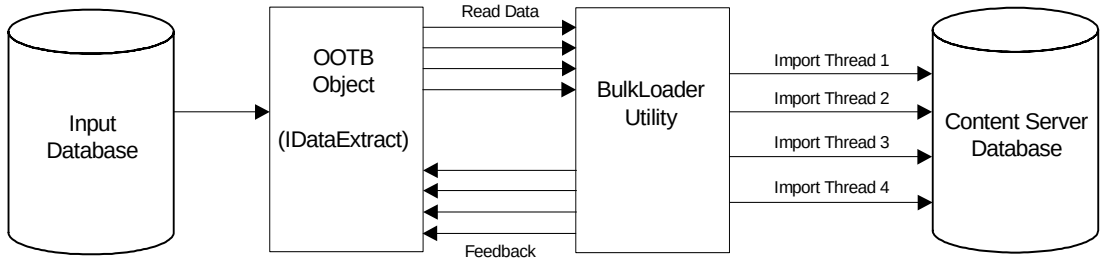


Figure 7: Built-in OOTB ("out of the box") implementation of BulkLoader

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 389.

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 423.

Inserts

The name of the data source table is specified by the `inputTable` parameter in the configuration file.

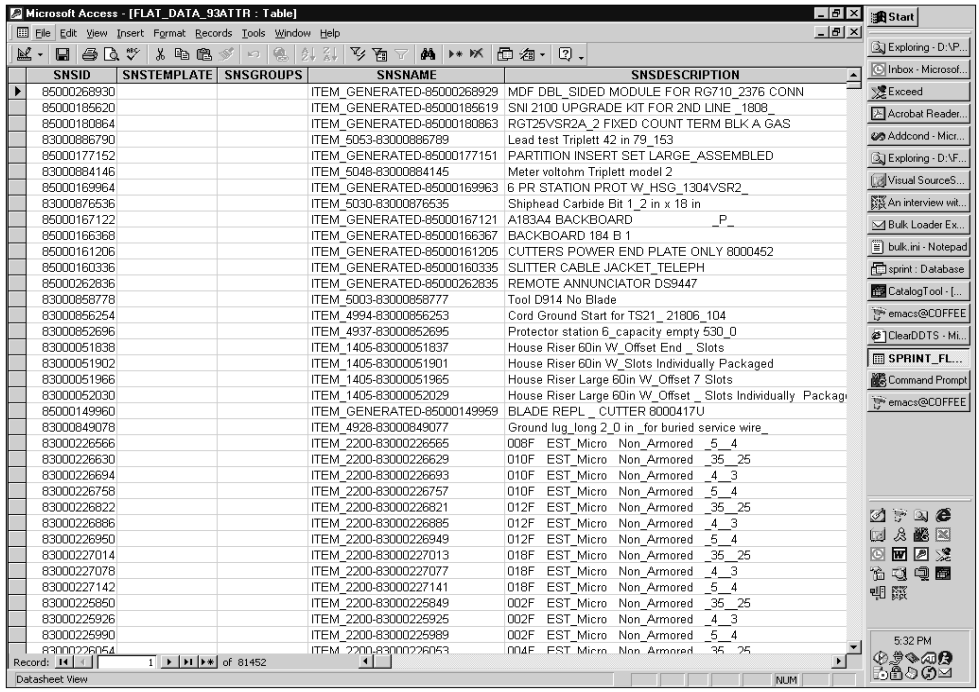
The source table must also include the names of the following four columns, which you specify in the configuration file with the following properties:

- `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):



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:

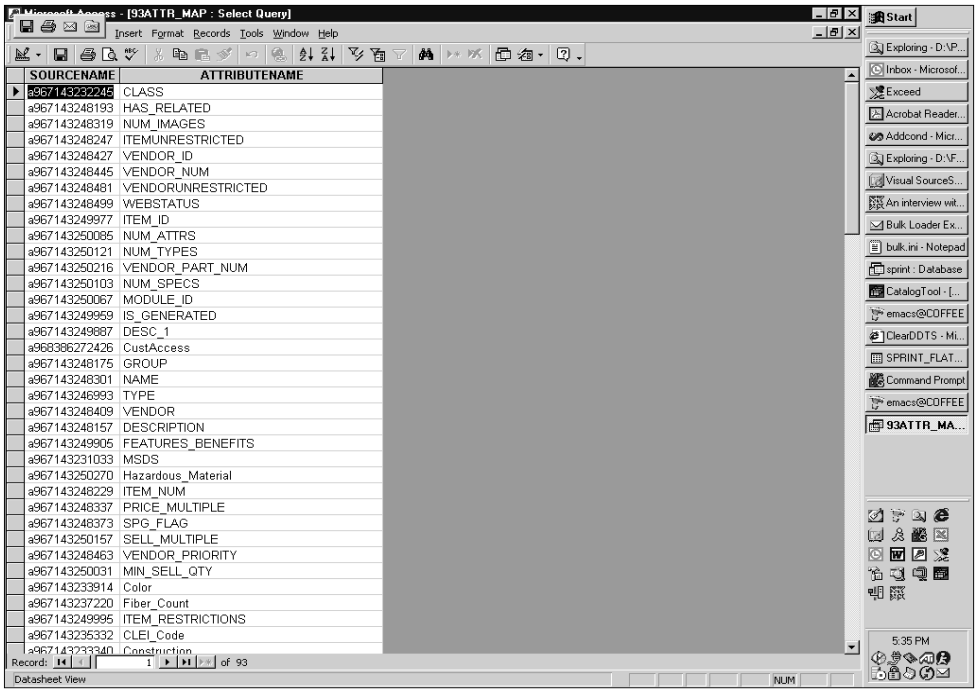
- A column that holds the names of the flex attribute columns in your flat data source
- A column that holds their corresponding names in the Content Server database

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:



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. Example: `4`
`dataSliceSize`	Required	Number of items retrieved in one read request; this number will also be processed by a single processing thread. 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. Default value ("out of the box"): `com.openmarket.gatorbulk.DataExtractImp`
`initId`	Required	Starting Content Server ID used the very first time BulkLoader operates; subsequently will use the value from `idSyncFile`. Example: `800000000000`
`idSyncFile`	Required	Next available Content Server ID is saved in this file; updated during a BulkLoader session Example: `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). Example: `1000`
`commitFrequency`	Required	Number of flex asset groups to be part of a database transaction. 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. Example: `oracle.jdbc.driver.OracleDriver`
`outputJdbcURL`	Required	The JDBC URL. The following example value is a typical type2 oracle JDBC driver URL: `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. For example: `inputAttributeMapTableKeyCol=SOURCENAME`
`inputAttributeMapTableValCol`	Required	The name of the column in the mapping table that lists the corresponding attribute names. For example: `inputAttributeMapTableValCol=ATTRIBUTENAME`
`inputTableDescriptionColumn`	Required	The name of the column in the source table that contains the descriptions of the flex assets. For example: `inputTableDescriptionColumn=SNSDESCRIPTION`
`inputTableGroupsColumn`	Required	Name of the column in the source table that contains the names of parents. Each value can include several parents, separated by the `multivalueDelimeter` character, which is defined in the configuration file. For example: `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. For example: `inputTableNameColumn=SNSNAME`
`inputTableTemplateColumn`	Required	The name of the column in the source table that contains the flex definitions. For example: `inputTableTemplateColumn=SNSTEMPLATE`
`createdby`	Required	The user name that you want to be entered in the `createdby` field for your flex assets. For example: `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. For more information, see ““Setting the initID Parameter” on page 429.
`multivalueDelimeter`	Required	The delimiter that separates multiple attribute values. The default character is the semicolon (**;**). For example: `multivalueDelimiter=;`
`siteName`	Required	The name of the site. All products will behave as if they were created under this site. For example: `siteName=GE Lighting`
`status`	Required	The status code for all imported flex assets. You should set this to `PL` for imported. For example: `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. 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. 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`. 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 444.

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. Example: `4`
`dataSliceSize`	Required	Number of items retrieved in one read request; this number will also be processed by a single processing thread. 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. Default value (“out of the box”): `com.openmarket.gatorbulk.DataExtractImp`
`initId`	Required	Starting Content Server ID used the very first time BulkLoader operates; subsequently will use the value from `idSyncFile`. Example: `800000000000`
`idSyncFile`	Required	Next available Content Server ID is saved in this file; updated during a BulkLoader session Example: `C:\FutureTense\BulkLoaderId.txt`

Property Name	Required/ Optional	Comments
`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). Example: `1000`
`commitFrequency`	Required	Required. 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. For example: `commitFrequency=5` (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. Example: `oracle.jdbc.driver.OracleDriver`
`outputJdbcURL`	Required	The JDBC URL. The following example value is a typical type2 oracle JDBC driver URL: `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 using BulkLoader.

BulkApprover can do the following tasks:

- Notify the approval system of all updates and deletions done 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 publishing target, without actually publishing them.

Note that only users with the `xceladmin` role can run BulkApprover.

Creating a Configuration File

Before you run BulkApprover for the first time, you must create a configuration file for the utility. You can create a separate `BulkApprover.ini` file for this purpose, or you can append the BulkApprover configuration information to one of BulkLoader's `.ini` files.

The following table lists the configuration information that you must provide:

Parameter	Description
`bulkApprovalURL` (Required)	The URL on the host server that has the data imported with BulkLoader. The correct value is as follows: `http://myServer/cs/ContentServer?pagename=OpenMarket/Xcelerate/Actions/BulkApproval` where *`myServer`* is the name of the host server.
`adminUserName` (Required)	The Content Server username of a user with the `xceladmin` role.
`adminUserPassword` (Required)	The password of a user with the `xceladmin` role password.
`approvalTargetList` (Required)	A list of the destinations that the assets are to be approved for. Separate each destination with the delimiter that you specify in the `multiValueDelimiter` parameter. For the names of destinations, see the Publish option on the Admin tab, or the name column of the `pubtarget` table. The syntax is: *`name1<multiValueDelimiter>name2<multiValueDelimiter>name3`*

Parameter	Description
`multiValueDelimiter` (Required)	A delimiter that you select. You use this delimiter to separate the approval targets that you specify in the `appovalTargetList` parameter.

`assetIdSqlFilter` (Optional)	A statement that can be appended to a SQL `WHERE` clause in order to filter asset IDs.
	For example:
	`asset_id%20=0`
	or
	`asset_id%20!=0`

`debug` (Optional)	Turns BulkApprover's debugging on and off. A value of `true` turns debugging on. Leave this parameter blank for no debugging. Debug messages are written to the file specified in the `output_file` parameter of the command line.
`assetschunksize` (Optional)	Specifies the number of assets that are approved in a single transaction. For example, setting this property to 20 means that the assets get approved in groups of 20. Setting this property helps prevent session timeouts. Default value: `25`
`outputJdbcDriver` (Required)	The name of the JDBC driver class to access the Content Server database. Example: `oracle.jdbc.driver.OracleDriver`
`outputJdbcURL` (Required)	The JDBC URL. The following example value is a typical type 2 oracle JDBC driver URL: `Jdbc:oracle:oci8:@foo`
`outputJdbcUsername` (Required)	Content Server database user name
`outputJdbcPassword` (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 and initialized the BulkApprover utility, you can use it to approve assets that you imported into the database using the BulkLoader utility.

BulkApprover accepts several parameters, which are described in the following table:

Parameter	Description
`config`	The name of the file where your BulkApprover configuration information is located; for example, `BulkApprover.ini`.
`action`	The action or actions that you want BulkApprover to perform. When you want BulkApprover to perform multiple actions, supply the values in a comma-separated list. Note that none of these values are required. Valid values are: • `notify` - Notifies the approval system about all updates and voids processed during a previous BulkLoader session. • `approve` - Tells BulkApprover to approve all of the assets that it processes for a given publishing destination(s). • `mark_publish` - Marks all of the assets that it processes as published on a given publishing destination(s), without actually publishing them. You specify the publishing targets using the `approvalTargetList` parameter found in the BulkApprover configuration file. 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`.

BulkApprover runs from the command line. To run the utility, set the 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
```

When you use BulkApprover to approve flex assets that you have loaded using BulkLoader, you must supply at least the `notify` or `approve` value for the `action` parameter.

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 20, “Creating Template, CSElement, and SiteEntry Assets”
- Chapter 21, “Creating Templates with InSite Templating”
- Chapter 22, “Creating Collection, Query, Stylesheet, and Page Assets”
- Chapter 23, “Coding Elements for Templates and CSElements”
- Chapter 24, “Template Element Examples for Basic Assets”
- Chapter 25, “User Management on the Delivery System”
- Chapter 26, “The HelloAssetWorld Sample Site”
- Chapter 27, “The Burlington Financial Sample Site”

Chapter 20

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 elements and page names, 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 23, “Coding Elements for Templates and CSElements.”

This chapter contains the following sections:

- Pages, Pagelets, and Elements
- CSElement, Template, and SiteEntry Assets
- Creating CSElement Assets
- Creating Template Assets
- Managing Template, CSElement, and SiteEntry Assets

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—that is, a URL—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 identify and then load assets to display in those pages or pagelets, and pass 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.

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/ContentServer?pagename=name_of_page`

- For iPlanet Application Server (iAS):

  `http://host:port/NASApp/cs/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, 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://host:port/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, you use the `render:satellitepage` (`RENDER.SATELLITEPAGE`) tag. For example:

```
<render:satellitepage pagename="BurlingtonFinancial/Page/Home"/>
```

To call an element directly by name, you use the `render:callelement` (`RENDER.CALLELEMENT`) tag. For example:

```
<render:callelement elementname="BurlingtonFinancial/Common/
TextOnlyLink"/>
```

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 tag although typically you call a template by its page name.

Page vs. Pagelet

For the sake of clarity, the following table summarizes the various definitions of the word "page" as used in the documentation for Content Server and the CS modules and 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. A pagelet is cached in the Content Server and Satellite Server page caches.

Term	Definition
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. A page is 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	A site design asset that uses the basic asset data model. You create page assets and then place them in position 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 Page assets do **not** represent page names.

CSElement, Template, and SiteEntry Assets

As mentioned, CSElement assets are elements, SiteEntry assets are page names, and Template assets are 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

Never use the revision tracking feature in the CS-Explorer tool to enable revision tracking directly on the `SiteCatalog` or `ElementCatalog` tables.

When you create a CSElement or Template asset, you create an element. You can code the element by using the asset form (CSElement or template, depending on which type of asset you are creating) or by using the CS-Explorer utility.

It is preferable that you at least start coding the element in the asset form before continuing to code it in CS-Explorer. The Content Server interface will seed your element with stub code that sets compositional dependencies and, if you are using JSP, drops in the appropriate tag library directives for you. (Note that compositional dependencies are described in the section “About Dependencies” on page 508.)

If you edit an element in the CS-Explorer tool, be sure to open and save the template or CSElement in the Content Server interface so that the approval system knows that the asset was changed and can change its approval status, and so the CacheManager servlet can update the cache.

Note

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.

Template Assets

Templates create the look and feel of your online site and render other assets into pages and pagelets. 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 it without 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.
- If you have coded the element in the template form, 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.

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 34, "Recommendation Assets."

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.

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, it 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 name of the page entry uses the following convention:

  ```
  SiteName/AssetTypeName/TemplateName
  ```

  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.

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. In this case 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.

Creating CSElement Assets

When you create a CSElement asset, you do two things: you create an asset, and you code an element for the asset. The fields in the first part of the "CSElement" form describe the element as an asset. The rest of the fields provide information about the element file, information that is written to the `ElementCatalog` table, and provide an interface that you can use to code the element.

This procedure describes how you create the CSElement asset. For help with coding the element, see Chapter 23, "Coding Elements for Templates and CSElements."

Before You Begin

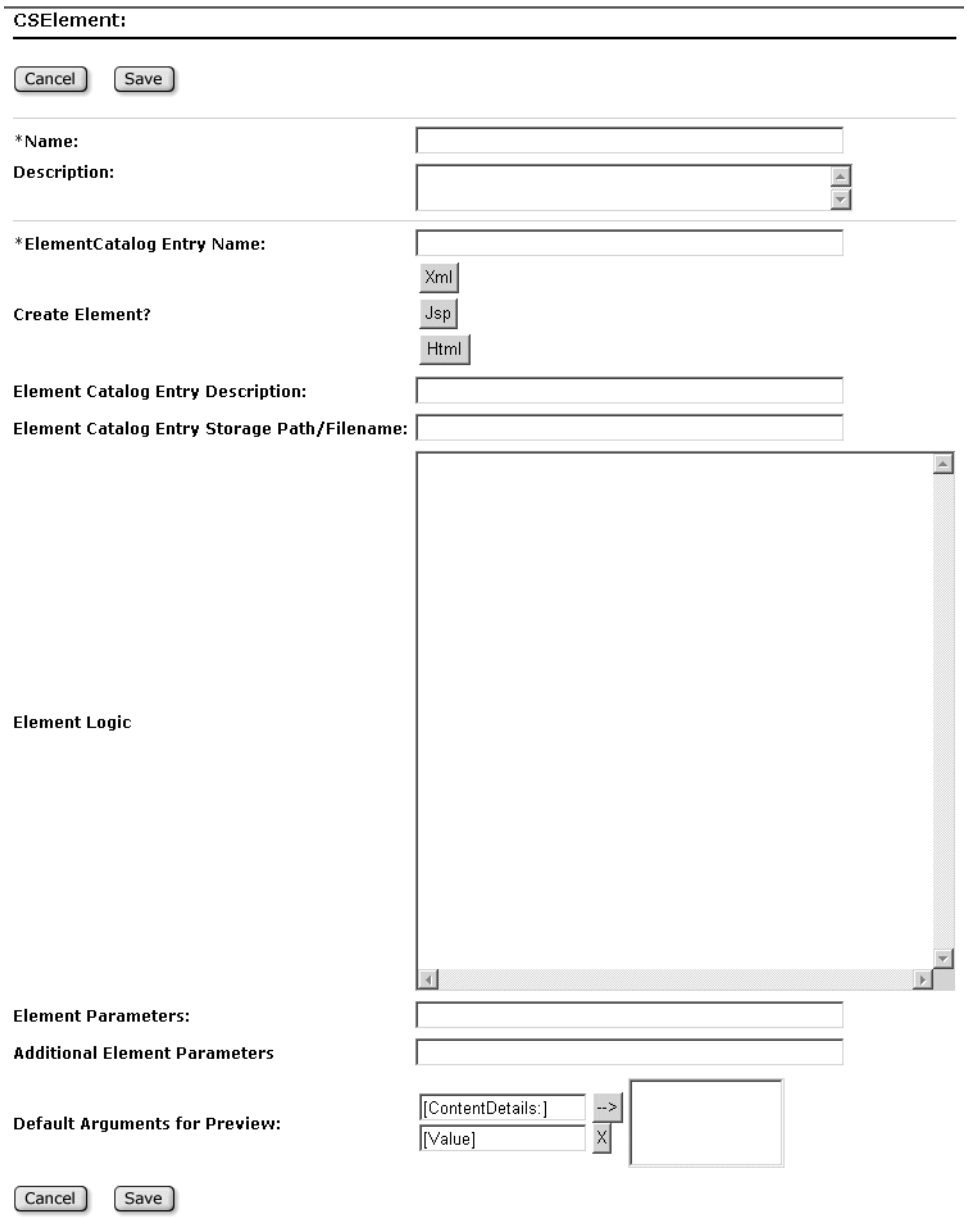
Before you begin, you must determine the naming convention that you want to use for your CSElement assets because you cannot change the names of an element after you save the asset.

Step 1: Name and Describe the CSElement Asset

Complete the following steps:

1. If necessary, log in to the Content Server interface.
2. Click **New** on the button bar.
3. Select **CSElement** from the list of asset types. (The CSElement asset type must be enabled for your site and there must be a start menu item created for it.)

The “CSElement” form appears:



4. (Required) In the **Name** field, type a descriptive name for the CSElement asset. It’s best to use a name that describes what the element does.
5. In the **Description** field, type a brief description of the CSElement asset. You can enter up to 128 characters.

Step 2: Configure the Element

The rest of the fields in the "CSElement" form define the element entry in the `ElementCatalog` table and provide the code for the element file. You can use either this section of the form or the Content Server Explorer tool to actually code the CSElement.

You should use this form to begin coding your element because it seeds the element with standard stub code that you need to include in every one of your elements. CS-Direct adds JSP `taglib` statements (only if you are using JSP) and the `RENDER.LOGDEP` tag to the file by default so that the compositional dependency between this CSElement asset and pages that are rendered from this element is logged. When you use CS-Explorer to create a new element, it does not drop in the `taglib` statements for assets or log the compositional dependency.

This section describes how to use the "CSElement" form only; it does not describe how to code an element in CS-Explorer.

For information about dependencies, see "About Dependencies" on page 508.

Option 1: Coding a New Element

If you are creating a new element, complete these steps:

1. In the **Create Element** section, 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.

   Content Server does the following:

   - Enters header and other information in the **Element Logic** text entry area for you.

     For example, it sets the beginning and ending `FTCS` tags, and it sets a `RENDER.LOGDEP` (`render:logdep`) tag to mark a dependency between the CSElement asset and any page or pagelet rendered with the element.

     If you clicked the **JSP** button, CS-Direct also sets a tag library directive for each of the CS-Direct JSP tag libraries (asset, siteplan, render).
   - Enters a name for the element file in the **ElementCatalog Entry Name** field. It uses the name of the CSElement asset by default.
   - Enters an upload file name for the element in the **ElementCatalog Entry Storage Path/Filename.** (This is the value of the `url` column in the element entry row in the `ElementCatalog` table).

     It uses the name of the CSElement asset by default.
2. Click in the **ElementCatalog Entry Name** field and edit the text so that it represents the "location" in the `ElementCatalog` table that you want. Note that the value in this field is the name of the element. When you create code that calls this element (`RENDER.CALLELEMENT`), this is the name you should use.

Use **forward** slashes to delineate the folders in the `ElementCatalog` table. For example: BurlingtonFinancial/Common/AlternateNavBar. If the folders you specify do not yet exist in the `ElementCatalog` table, Content Server creates them when you save the CSElement asset.

3. Click in the **ElementCatalog Entry Storage Path/Filename** field and edit the text so that it matches the text that you entered in the **ElementCatalog Entry Name** field**.**
4. In the **ElementCatalog Entry Description** field, enter a description of the element. When you save the CSElement asset, the information in this field is written to the `description` column for the element entry in the `ElementCatalog` table.
5. In the **Element Logic** entry area, code your element. Be sure to enter all of your code before the ending `ftcs` tag.

   If you are using JSP, be sure to remove the comments from the `taglib` directives that describe the tag families that you are using.

   If you want, you can cut and paste code that you have already written. Or, you can use Content Server Explorer to continue coding this element **after** you **save** the asset. If you decide to use Content Server Explorer, complete the rest of the steps in this procedure first—before you use Content Server Explorer to open the element.

   For help with this step, see Chapter 23, "Coding Elements for Templates and CSElements."
6. (Optional) If your site design requires it, enter the appropriate arguments in the element parameter fields:
   - The **Element parameters** field corresponds to the `resdetails1` column for the element entry in the `ElementCatalog` table.
   - The **Additional element parameters** field corresponds to the `resdetails2` column for the element entry in the `ElementCatalog`.

   These columns hold variables or arguments that can be passed to the element. If your site design requires that you use additional variables for your elements, enter them as name/value pairs with multiple arguments separated by the ampersand (&) character. For example:

   ```
   MyArgument=value1&YourArgument=value2
   ```

   By default, CS-Direct uses the `resdetails1` column to set the `eid` variable to the value of the CSElement asset's ID, but it does not use the `resdetails2` column.

   Note

   If you work with the element in CS-Explorer, be sure that you do not alter the value of the `eid` variable or accidentally delete it.

   For more information about Content Server variables, including scope and precedence, see Chapter 4, "Programming with Content Server."
7. For information about the **Default Arguments for Preview** field, see "Previewing Template, CSElement, and SiteEntry Assets" on page 473.

Option 2: Selecting an Element that Already Exists

If you are creating a CSElement for an element that already exists, you just need to identify the element correctly and then CS-Direct can find it and associate it with this CSElement asset.

Finish creating the asset by completing the following steps:

1. In the **ElementCatalog Entry Description** field, type a description of the element. When you save the CSElement asset, the information in this field is written to the `description` column for the element entry in the `ElementCatalog` table.

Note

Because the naming convention requires element names to be unique, you should not make two or more Template assets point to the same element. You can, however, make two SiteEntry assets point to the same element.

2. Complete step 4 from "Option 1: Coding a New Element" on page 468.

When you save the CSElement asset, CS-Direct checks for the presence of an element with the correct name:

- If the element already exists, CS-Direct recognizes it and the next time you edit the CSElement, the code from the element is displayed in the **Element Logic** field.
- If the element does not yet exist, 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.

For help with coding your template elements, see Chapter 23, "Coding Elements for Templates and CSElements."

Step 3: Save and Inspect the CSElement

When you are finished creating the CSElement asset, click **Save**.

CS-Direct does the following:

- Creates a row in the `CSElement` table for the CSElement asset.
- Creates an entry in the `ElementCatalog` table using names specified in the form:
  - The value from the **ElementCatalog Entry Name** field is used to position the element file in the appropriate folder.
  - The value from the **ElementCatalog Entry Storage Path/Filename** field is written to the `url` column.
  - Additionally, CS-Direct sets the value of the `eid` variable to the ID of the CSElement asset in the `resdetails1` column.

- Displays the “Inspect” form.

  For example:

  The “Inspect” form for a CSElement asset displays the following kinds of information:

  - The standard summary information displayed for assets of all types (asset name, description, status, ID, source, and so on).
  - The element entry information that you entered for the CSElement’s element and that CS-Direct set for you (that is, the value of the eid variable).
  - The code in the element itself.

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 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.

Before you begin, complete the following tasks:

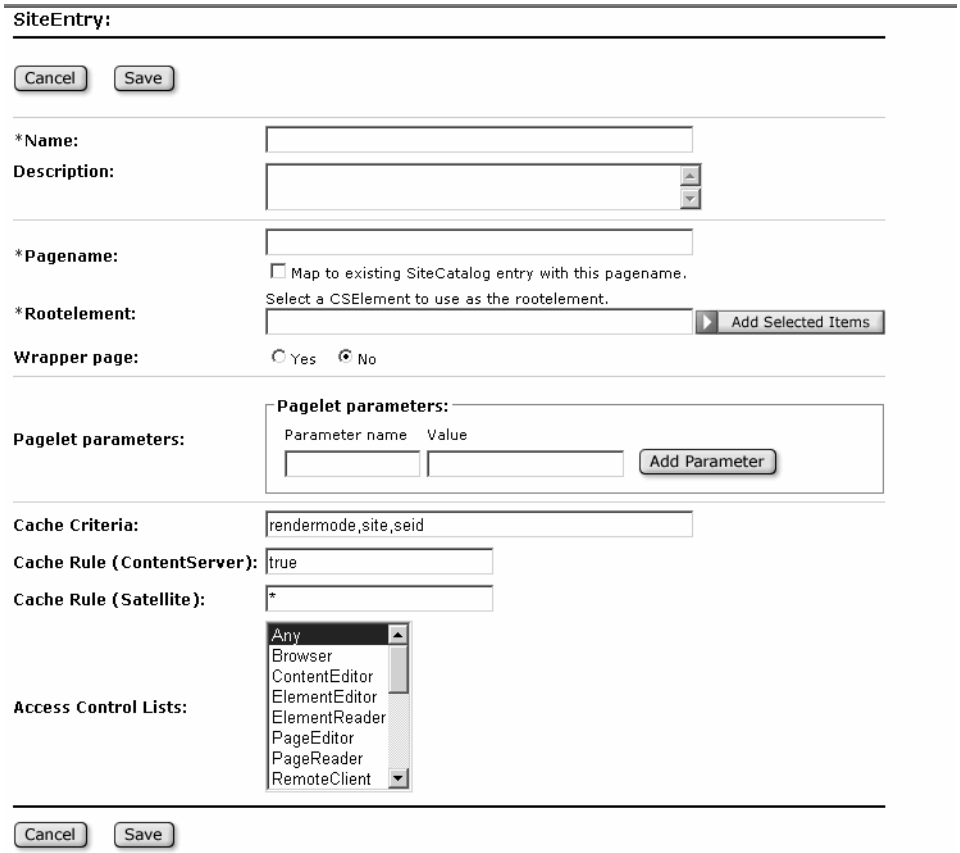
- Create a root element for the page entry.
  (A root element is required for any page entry. When you create a SiteEntry asset, you must select a CSElement asset that represents the root element.)
- Make your CSElement assets available from the Content Server tree.
  (You specify the CSElements for your SiteEntry assets by selecting them from either the “History” tab or the “Active List” tree tab.)
- Determine the naming conventions that you want to use, because you cannot change the name of a page entry after you save a SiteEntry asset.

To create a SiteEntry asset

1. If necessary, log in to the Content Server interface.
2. Select the site in which you want to work.
3. Click **New** on the button bar.
4. Select **New SiteEntry** from the list of asset types.

> **Note**
>
> For the **New SiteEntry** name to be listed, the SiteEntry asset type must be enabled for your site and there must be a start menu item created for it.

The “SiteEntry” form appears:



5. (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.
6. In the **Description** field, type a brief description of the SiteEntry asset. You can enter up to 128 characters.
7. (Required) In the **Pagename** field, enter the page name that you want stored in the `SiteCatalog` table. Note that the value in this field is the name of the page entry. When you create code that calls this SiteEntry asset (`RENDER.SATELLITEPAGE`), this is the name you should use.

   Use **forward** slashes to delineate the folders in the `SiteCatalog` table. For example: BurlingtonFinancial/Common/AlternateNavBar. If the folders you specify do not yet exist in the `SiteCatalog` table, Content Server creates them when you save the SiteEntry asset.
8. (Required) In the **Rootelement** field, select the appropriate CSElement asset from the tree and click **Add Selected Items**.
9. In the **Wrapper page** field, select one of the checkboxes to identify whether the asset you are creating is a wrapper page. The **No** checkbox identifies the asset to be a pagelet.

10. In the **Pagelet parameters** field, you can enter default values for variables that will be passed into the page or pagelet each time it is executed. These variables will be overwritten if they are explicitly set when the page or pagelet is called. If you are specifying a parameter here, its name must be listed as a cache criteria parameter below. The name-value pairs you enter here are stored in the resargs1 and resargs2 columns of the SiteCatalog (the column into which they are entered is not important and is managed automatically).
11. In the **Cache Criteria** field, enter a comma-separated list of all of the variables that will be passed into the page. The variables passed into the page will be used to compute the cache key that will uniquely identify the pages. In other words, the page or pagelets incoming parameters are filtered by the cache criteria and a cache key is constructed with the remaining variables. Parameters not specified by page criteria will not always be passed to the page when it is executed. The Cache Criteria value specified here is set as the pagecriteria column in the SiteCatalog.
12. In the **Cache Rule (ContentServer)** field, which corresponds to the `cscacheinfo` column in the `SiteCatalog` table, do one of the following:
    - Leave the field blank if you want the pagelet that is rendered by this SiteEntry asset's root element to be cached with the timeout value of the `cs.pgCacheTimeout` property in the `futuretense.ini` file. If you leave the field blank and save the asset, the default value is set to: `true,*`
    - Enter a value if you want to turn off caching or override the default timeout setting for the page entry for this SiteEntry asset. For example, to turn off caching for this pagelet, type `false` in the field. To override the default timeout value for this pagelet, type `true,~`*n* where *n* is the number of minutes the item should be kept in the cache before expiring.

    Because CacheManager can be relied upon to clear the cache when content on cached pages changes, the recommended value for this field is `true,~-1,` which means "cache the page forever."

    For more information about page caching settings, see Chapter 5, "Page Design and Caching."
13. In the **Cache Rule (Satellite)** field, which corresponds to the `sscacheinfo` column in the `SiteCatalog` table, do one of the following:
    - Leave the field blank if you want the pagelet that is rendered by this SiteEntry asset's root element to be cached with the timeout value of the `cs.pgeCacheTimeout` property in the `futuretense.ini` file. If you leave the field blank and save the asset, the default value is set to: `true,*`
    - Enter a value if you want to turn off caching or override the default timeout setting for the page entry for this SiteEntry asset. For example, to turn off caching for this pagelet, type `false` in the field. To override the default timeout value for this pagelet, type `true,~n` where `n` is the number of minutes the item should be kept in the cache before expiring.

      In no case should you set the **Cache Rule (Satellite)** field to a value that will result in the page residing in the cache on Satellite Server longer than it will reside on Content Server. Doing so will result in pagelets being cached that cannot be flushed.

Because CacheManager can be relied upon to clear the cache when content on cached pages changes, the recommended value for this field is `true,~-1`, which means "cache the page forever."

For more information about page caching settings, see Chapter 5, "Page Design and Caching."

14. 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 25, "User Management on the Delivery System.")
15. For information about the **Default Arguments for Preview** field, see "Previewing Template, CSElement, and SiteEntry Assets" on page 473.
16. Click **Save**.

Creating Template Assets

When you create a Template asset, you create it for a specific asset type. You provide formatting code for assets of that type and you can specify caching and other criteria for Content Server to use when rendering an asset that uses the template.

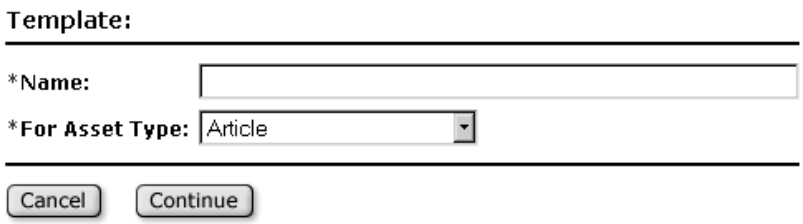
The procedure in this section describes the basic steps of creating and saving a Template asset. Coding samples and other information about coding templates for specific asset types follows in Chapter 23, "Coding Elements for Templates and CSElements."

Step 1: Name and Describe the Template Asset

Complete the following steps:

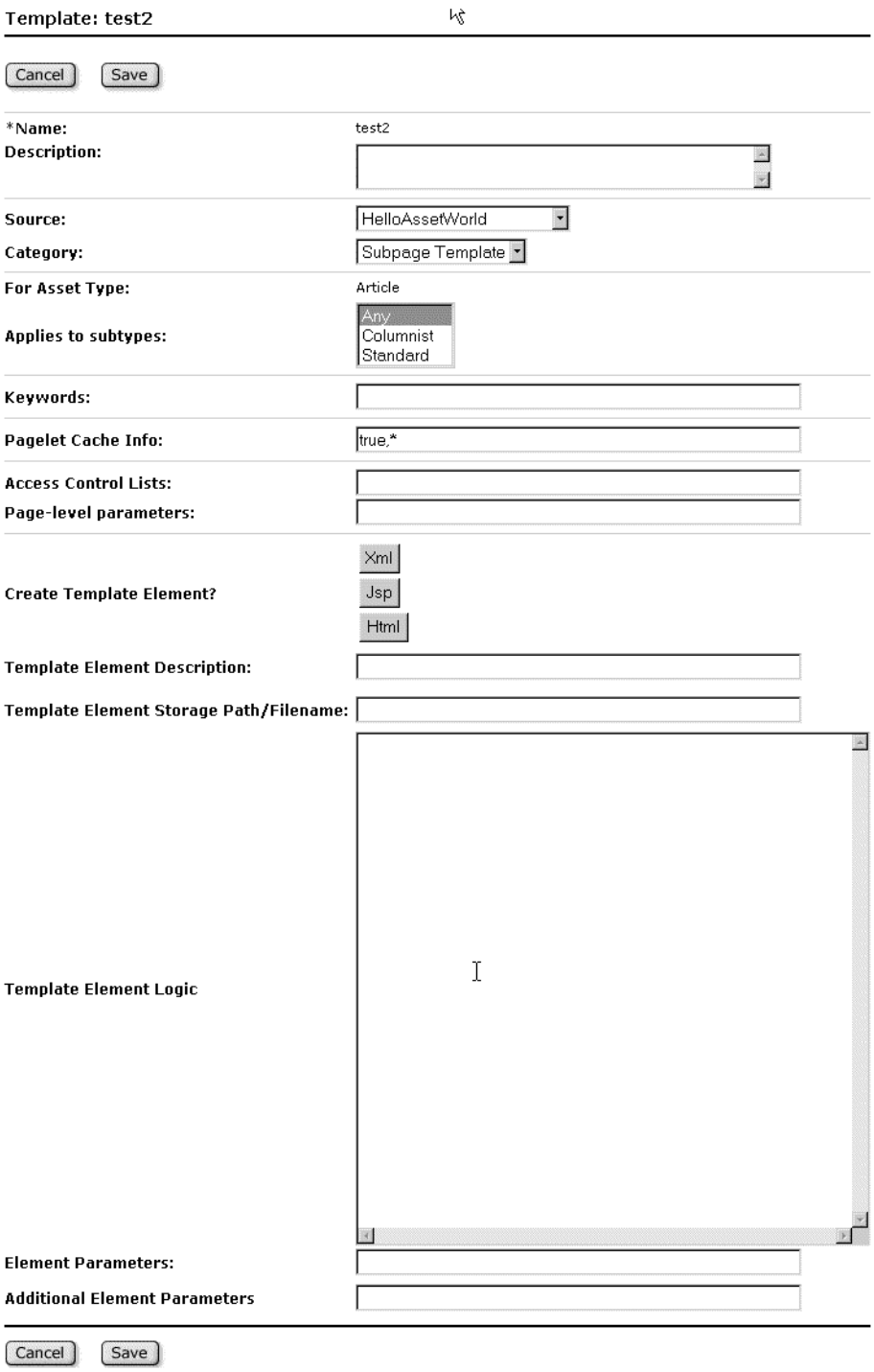
1. If necessary, log in to the Content Server interface.
2. Click **New** on the button bar.
3. Select **Template** from the list of asset types. (The Template asset type must be enabled for your site and there must be a start menu item created for it.)

   The first **Template** form appears:



4. (Required) In the **Name** field, type a descriptive name for the template that is unique for the type of asset that it formats. You can enter up to 64 alphanumeric characters, but the first character must be a letter. Underscores (_) and hyphens (-) are acceptable. Spaces are also acceptable; however, they will be converted to underscores (_) when used in the `SiteCatalog` pagename for the template.
5. (Required) Click in the **For Asset Type** field and select the asset type that this template formats. For example, if you are creating an article asset template, select **Article** from the drop-down list.

6. Click **Continue**. The following form appears:



The top part of the form describes the template as an asset.

7. In the **Description** field, type a brief description of the template. You can enter up to 128 characters.

8. If you are using the **Source** field, select a source from the drop-down list.
9. If you are using the **Category** field, select a category from the drop-down list.
10. If this template should be used only for specific subtypes of the asset type that you selected in **For Asset Type** field in step 5, select the appropriate subtypes in the **Applies to Subtypes** field.
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 User's Guide*.

Step 2: Configure the Page Name

The next three fields define the page entry in the `SiteCatalog` table for the Template asset:

1. (Optional) The **Pagelet Cache Info** field corresponds to the `cacheinfo` column in the `SiteCatalog` table. When you save the asset without entering information in this field, the default value is set to:

   `true,*`

   This means means that the pagelet rendered by this template's element will be cached and the timeout value is set to the value of the `cs.pageCacheTimeout` property in the `futuretense.ini` file.

   To override these settings for the page entry for this template, click in the field and enter the new values. For example, to turn caching off for this pagelet, type `false` in the field. To override the default timeout value for this pagelet, type `true,`*`n`* where *n* is the number of minutes the item should be kept in the cache before expiring.

   For more information about page caching settings, see Chapter 5, "Page Design and Caching."
2. (Optional) The **Access Control Lists** field corresponds to the `acl` column in the `SiteCatalog` table. If you want to restrict the visitors who can request this page, enter the ACLs that visitors must have in order to see the page. (For more information about ACLs, see Chapter 25, "User Management on the Delivery System.")
3. (Optional) The **Page Level Parameters** field corresponds to the `resargs2` column in the `SiteCatalog` table, a column that holds variables (arguments) that can be passed to the ContentServer with a page name. When you save the template, CS-Direct sets the following variables as PageCriteria variables in the `resargs1` column: `c`, `cid`, `p`, and `rendermode`. (For information about these variables, see Chapter 4, "Programming with Content Server.")

   The caching system uses only those variables that are specified as as PageCriteria variables to create the cachekey for cached pages. Therefore, if your site design requires that you use additional page-level variables, be sure to designate them as PageCriteria variables. Because the variables entered in this field (`resargs2` column) overwrite the variables that CS-Direct sets in the `resargs1` column, you must include the CS-Direct variables in your statement.

   For example, say that you want to include a variable named `foo` as a PageCriteria variable. In this case, you would enter the following string in this field:

   `PageCriteria=c,cid,p,rendermode,foo`

Step 3: Configure the Element

The rest of the fields in the template form define the element entry in the `ElementCatalog` table. You can use either this section of the form or the Content Server Explorer tool to actually code the template. For help with coding the element, see Chapter 23, "Coding Elements for Templates and CSElements."

When you use this section of the form to code or to start coding the element, CS-Direct adds JSP `taglib` statements (only if you are using JSP) and the `RENDER.LOGDEP` tag to the file by default so that the compositional dependency between this Template asset and pages that are rendered from this element is logged. For information about dependencies, see "About Dependencies" on page 508.

Option 1: Coding a New Element

If you are creating a new element for a new Template asset, complete these steps:

1. In the **Create Template Element?** section, do one of the following:
   - To create an .xml file, click **XML**.
   - To create a .jsp file, click **JSP**
   - To create an .html file, click **HTML**.

   CS-Direct does the following:

   - Enters header and other information in the **Template Element Logic** text entry area for you.

     For example, it sets a `RENDER.LOGDEP` (`render:logdep`) tag to mark a dependency between the Template asset and any page or pagelet rendered with the template.

     If you clicked the **JSP** button, CS-Direct enters a tag library directive for each of the CS-Direct JSP tag libraries.
   - Enters an upload file name for the element in the **Template Element Storage Path/Filename.** (This is the value of the `url` column in the element entry row in the `ElementCatalog` table).

     CS-Direct uses the *`AssetTypeName/TemplateName`* naming convention.

     For example, if you named the template Summary and the asset type is article, the upload file name for the template element is called `Article/Summary` in the `ElementCatalog` table.
2. (Optional) In the **Template Element Description** field, enter a description of the template. When you save the Template asset, the information in this field is written to the `description` column for the element entry in the `ElementCatalog` table.
3. In the **Template Element Logic** entry area, code your element. If you are using JSP, be sure to remove the comments from the `taglib` directives that describe the tag libraries that you are using.

   If you want, you can cut and paste code that you have already written. Or, you can use Content Server Explorer to continue coding this template **after** you have **saved** the asset. If you decide to use Content Server Explorer, complete the rest of the steps in this procedure first, before you use Content Server Explorer to open the element.

   For help with this step, see Chapter 23, "Coding Elements for Templates and CSElements."

4. The **Element parameters** field corresponds to the `resdetails1` column and the **Additional element parameters** field corresponds to the `resdetails2` column for the element entry in the `ElementCatalog`. These columns hold variables or arguments that can be passed to the element.

   When you save the asset, CS-Direct sets a value for the `tid` variable in `resdetails1`. If your site design requires that you use additional variables for your template elements, enter them in these fields. For more information about using variables, see Chapter 4, "Programming with Content Server."

Option 2: Selecting an Element that Already Exists

If you are creating a Template asset for an element that already exists, you just need to identify the element correctly so that CS-Direct can find it and associate it with this Template asset.

As long as you use the proper naming convention for your template element—*`AssetTypeName/TemplateName`*—CS-Direct can find it and associate it with the appropriate template.

Complete the following steps:

1. (Optional) In the **Template Element Description** field, type a description of the template.
2. Complete step 4 from "Option 1: Coding a New Element" on page 468.

When you save the Template asset, CS-Direct checks for the presence of an element with the correct name:

- If the element already exists, CS-Direct recognizes it and the next time you edit the template, the code from the element is displayed in the **Template Element Logic** field.
- If the element does not yet exist, 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.

For help with coding your template elements, see Chapter 23, "Coding Elements for Templates and CSElements."

Step 4: Save and Inspect the Template

When you are finished creating the Template asset, click **Save**.

CS-Direct does the following:

- Creates a row in the `Template` table for the Template asset.
- Determines the name of the site that the template belongs to.
- Creates a page entry in the `SiteCatalog` table using the *`SiteName/AssetTypeName/TemplateName`* naming convention. When you create code that calls this Template asset (`render:satellitepage`), this is the page name that you use.
- Sets the name of the root element of the new `SiteCatalog` page entry to the name of the `ElementCatalog` entry (or to what it expects will be the name of the root element).

- Creates an entry in the `ElementCatalog` table using the *`AssetTypeName/TemplateName`* naming convention. If the element was coded in the template form, it also creates the element file.
- Displays the **Inspect** form.

For example:

The **Inspect** form for a Template asset displays the following kinds of information:

- The standard summary information displayed for assets of all types (asset name, description, status, ID, source, and so on).
- The page entry and element entry information that you entered for the template and its element.
- The following additional information from the page entry in the `SiteCatalog` table information that you are not allowed to edit from the template form:
  - The value from `csstatus`, a `SiteCatalog` column that determines whether the page can be viewed in a browser. CS-Direct sets `csstatus` to `live`, which indicates yes.
  - The value from `resargs1`, the `SiteCatalog` column that holds the default variables that CS-Direct sets for each template.

    For all Template assets, the first part of this statement is `PageCriteria=cid,ct,p,rendermode&rendermode=live`

    Additionally, CS-Direct sets the value for `c` and `tid` for the template. The `tid` variable is set to the ID of the Template asset. The `c` variable is set to the value of the asset type selected for the template

For example, if the Template asset is for article assets and its ID is 1012950138609, the `resargs1` column is set to:

```
PageCriteria=c,cid,ct,p,rendermode&rendermode=live&
c=Article&tid=1012950138609
```

For more information about these variables, see Chapter 4, "Programming with Content Server."

- The code in the template's element.

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.

Managing Template, CSElement, and SiteEntry Assets

This section presents additional procedures for working with template, CSElement, and SiteEntry assets:

- Designating Default Approval Templates (Export to Disk 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 (Export to Disk 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 510. 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 9: (Optional) Configure Subtypes" on page 300).

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.

Note

You also cannot change the name of a site after it is created.

For the basic procedure for editing assets, see the *Content Server 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 User's Guide*.

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 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 than you preview your content assets.

Templates and Preview

When you preview a template, you preview it in the context of an asset. You preview the asset and select 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

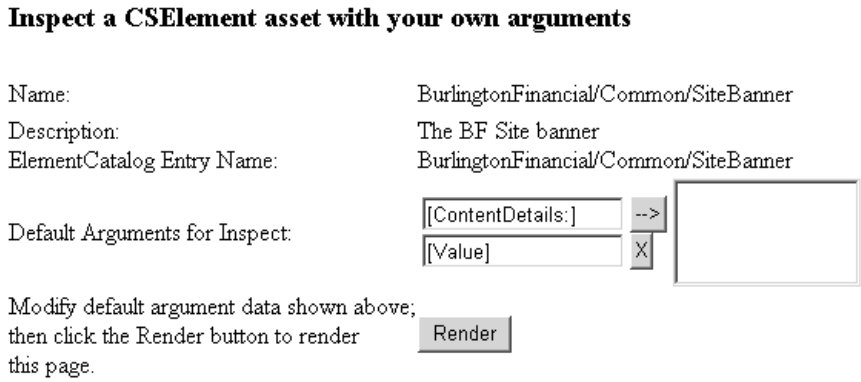
You preview CSElement and SiteEntry assets directly. If the element that will be called has self-contained context—a banner that does not expect variables or arguments, for example—you can simply click the **Preview** icon. But when the results of the rendered element will depend on values that are passed 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 status form. Next to the **Default Arguments for Preview** field, click **View**.

   This is the form that appears for a CSElement asset:



3. Click in the top field next to the words **Default Arguments for Inspect** and enter the name of the argument or variable.
4. Click in the lower field and enter the value for the argument or variable that you specified.
5. Click the arrow button.

   The name/value pair is moved to the text box on the right.
6. Repeat steps 3 through 5 for each argument that your asset needs a value for in order to preview it.
7. Click **Render** to preview the asset.

Note that you can provide and save preview arguments for your SiteEntry and CSElement assets with the **Default preview arguments** field in their “New” and “Edit” forms.

Chapter 21

Creating Templates with InSite Templating

The InSite Templating feature enables content providers to graphically create and redesign web pages by using an editorial feature called a “master page.” A master page is the preview of a master template, which is coded to display placeholders for content. The placeholders are arranged in a certain layout (typically specified by a site designer). By previewing a master template with the required layout and dragging content into the placeholders, a business user can quickly create a new web page. The web page can just as easily be edited, without the business user having to manipulate the underlying template code.

Master templates are not out-of-the-box constructs. They must be specially coded to contain placeholders, which allow for changes to the layout and content of web pages.

InSite Templating complements rather than duplicates the function of the InSite Editor; whereas the InSite Editor allows content providers to edit the content itself, InSite Templating does not.

This chapter explains the concept of master templates and how to create the templates. This chapter contains the following sections:

- Overview
- Creating a Master Template
- Testing Master Templates

Overview

InSite Templating makes it easy for content providers to create and redesign web pages. Using the InSite Templating graphical interface, content providers simply drag content onto and out of the pages that are specially designed for editing. For example, if content must be inserted into a page header, the content can be searched for directly from the page, then dragged and dropped into the header. When the result is saved, the page is updated and displayed in its new layout on the site.

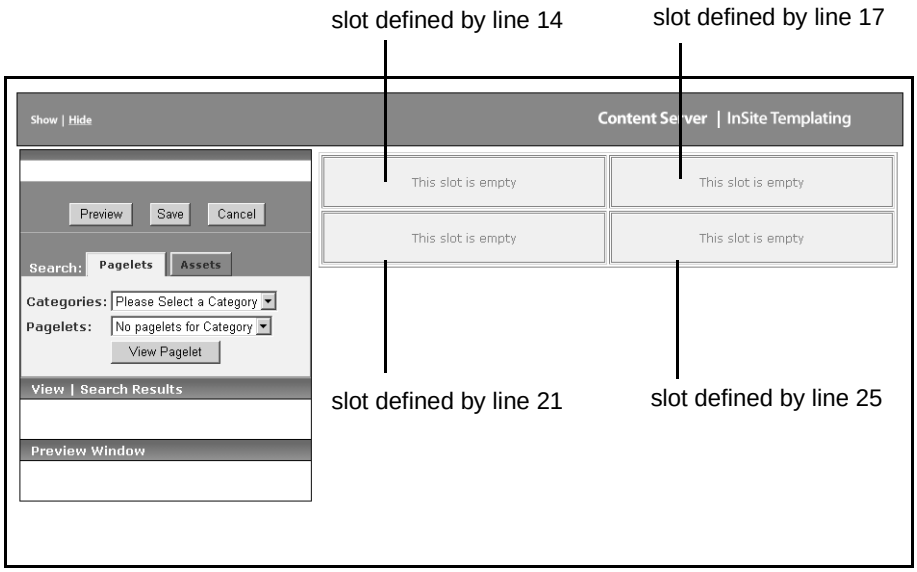
For developers, InSite Templating means coding JSP templates in a way that produces placeholders for content in the right places on the rendered page. Such templates are called **master templates**. The placeholders they contain are called **slots**, and the slotted pages they render are called **master pages**.

The process of setting up and using InSite Templating is summarized in the steps below.

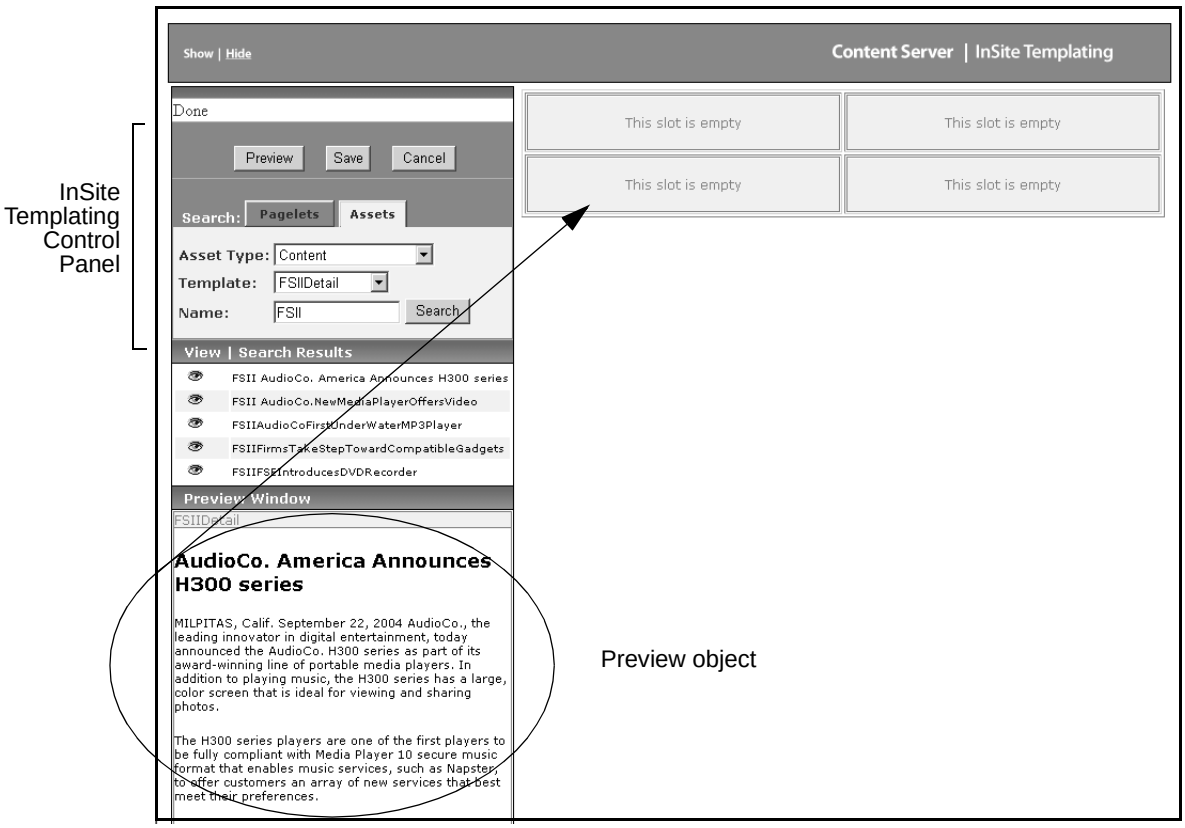
1. The developer writes a master template. For example, the following master template defines four slots (in lines 14, 17, 21, and 25):

```
1  <%@ taglib prefix="insitetemplating" uri="futuretense_cs/
   insitetemplating.tld"%>
2  <%@ page import="COM.FutureTense.Interfaces.*,
3                   COM.FutureTense.Util.ftMessage,
4                   COM.FutureTense.Util.ftErrors"%>
5  <cs:ftcs><%-- Page/MyTemplate--%>
6
7  <%-- Record dependencies for the Template --%>
8  <ics:if
   condition='<%=ics.GetVar("tid")!=null%>'><ics:then><render:logd
   ep cid='<%=ics.GetVar("tid")%>' c="Template"/></ics:then></
   ics:if>
9
10 <%-- Create slots --%>
11 <table border=1>
12 <tr>
13 <td>
14 <insitetemplating:slot slotname='row1col1'
   tid='<%=ics.GetVar("tid")%>' site='<%=ics.GetVar("site")%>' />
15 </td>
16 <td>
17 <insitetemplating:slot slotname='row1col2'
   tid='<%=ics.GetVar("tid")%>' site='<%=ics.GetVar("site")%>' />
18 </td>
19 </tr>
20 <td>
21 <insitetemplating:slot slotname='row2col1'
   tid='<%=ics.GetVar("tid")%>' site='<%=ics.GetVar("site")%>' />
22 </td>
23 <td>
24 <insitetemplating:slot slotname='row2col2'
   tid='<%=ics.GetVar("tid")%>' site='<%=ics.GetVar("site")%>' />
25 </td>
26 </tr>
27 </table
28 </cs:ftcs>
```

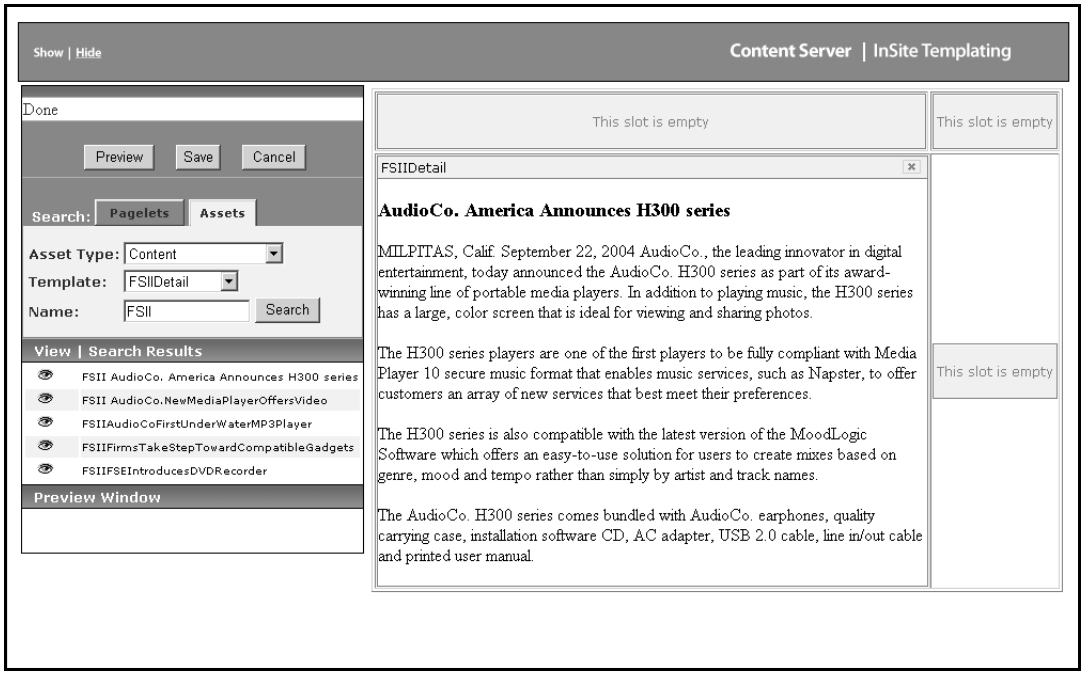
2. The content provider previews the template. The preview is called a “master page.”



3. The content provider defines and selects a preview object for a slot (using the InSite Templating control panel at the left of the page), then drags the preview object into the slot.



4. The result is shown in the figure below. Saving the result produces a publishable page.



Typically, master pages are drafted by site designers in several layouts to give content providers a choice of pre-approved page designs. Developers code the corresponding master templates by using the `<insitetemplating:slot>` tag in places where a slot should be displayed.

Designing Slots

Slots can be created only in a JSP template by inserting the following header line and tag:

- `<%@ taglib prefix="insitetemplating" uri="futuretense_cs/insitetemplating.tld"%>`

  The line above imports the library that enables you to use the `<insitetemplating:slot>` tag.
- The `<insitetemplating:slot>` tag creates a slot. The placement of the tag in the code determines the placement of the slot on the rendered page.

When designing slots, developers need to follow several rules:

- Slots can be created in both typed and typeless JSP templates.
- Slots must be named uniquely within the template code (otherwise, the master template cannot be saved).
- Slots must be created empty. They cannot refer to any asset.
- Slots cannot be superimposed on assets.
- Slots cannot be nested.
- Slots can be formatted. For example, slots in the context of a table can formatted to display borders of a specific width and color. (Note that according to best practices,

tables should not be used to specify page layout, but to present data in a tabular format.)

- If a master template refers to assets, those assets cannot occupy a slot. (The assets will still be rendered when the master template is previewed, but not within a slot. Such assets cannot be graphically manipulated on the master page.)

Procedures for creating slots in JSP templates are given in "Creating a Master Template," on page 481.

Choosing Content for Rendered Slots

Populating a slot with content requires the user to specify and select a preview object, using the InSite Templating control panel (see step 3 on page 477). The preview object can be:

- An asset-template pair. The template determines the form in which the asset is rendered (as the asset itself, a link to the asset, an index page referring to the asset, and so on).
- A pagelet. This is a special instance of a template that is coded with the `c` and `cid` parameters to display the required content (the c parameter is the type of asset to be rendered, and the `cid` parameter is the ID of an asset of that type). For the template to be listed as an option in the InSite Templating control panel, it must be assigned to a category whose name begins with `IT_` (case sensitive).

  A pagelet can also be a calling template. The pagelet must be assigned to a category beginning with `IT_` and pass the parameters that are needed for rendering.

The asset-template and the pagelet specifications are identical in terms of the preview object that is constructed, but not in terms of logic. The asset-template specification explicitly names the asset to be rendered and its rendering template. The pagelet specification names a pagelet, which carries information about the asset to be rendered and therefore determines which asset will be rendered in the slot.

Which specification to choose depends on individual preference. The pagelet specification simplifies InSite Templating for business users, as they are not required to know the specifics of asset types and rendering templates.

Placing Content

Placing content into a rendered slot requires dragging a preview object into the slot. Once content is placed, it can be moved from slot to slot on the master page, or removed from the master page.

- Content in a slot can be moved to another slot, replaced with different content, or deleted.
- At least one slot must be populated with content. Other slots can remain empty (depending on design considerations).
- Content that lies outside a slot is fixed on the master page. It cannot be repositioned, replaced, or removed except by recoding the master template.
- Content in multiple slots can be interchanged by dragging one piece of content over another.
- Slots cannot be nested.

- If users of a master template need to create multiple web pages from that template, they must first create a uniquely named copy of the master template for each web page they require, then modify the copy.

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). The next time that the master template is previewed, Content Server refers to the `Template_Composition` table and renders the master page with the content that is specified in the table.

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 that hosts the slot (`IST_Template`, in this example).
`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).
`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 (/). Note that the `cs_assetid` and the `cs_tname` form an asset-template pair (see "Choosing Content for Rendered Slots," on page 479).

The next time that the master template is previewed, Content Server refers to the `Template_Composition` table and renders the master page with the content specified in the table. **Note that the code of the master template remains unchanged.**

Creating a Master Template

A master template must be a JSP template. It is like any other JSP template that can be created in Content Server, except for the slots it contains. Slots are created by inserting the `<insitetemplating:slot>` tag into selected places, as shown in the steps below.

To create a master template

1. Create a new template, or open an existing template and use the following guidelines:
   a. In the "Name" screen:
      1) 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.
      2) Make sure the field "For Asset Type" correctly identifies the template as typed or typeless.
         - 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.
   b. In the "Element" screen:
      1) Make sure the selection in the "Usage" field is either **Usage unspecified** or **Element defines a whole HTML page**.
      2) In the "Create Template Element" field, select **JSP**.
   c. When coding the template:
      - Insert the following line into the header:

        ```
        <%@ taglib prefix="insitetemplating" uri="futuretense_cs/
           insitetemplating.tld"%>
        ```

        (The line above imports the library that enables you to use the `<insitetemplating:slot>` tag.)
      - Create slots in the template by inserting the tag `<insitetemplating:slot>` with the following parameters:
        - `slotname` (must be unique)
        - `tid` (the template's id)
        - `site` (the current site)

        For example:

        ```
        <insitetemplating:slot slotname='test'
           tid='<%=ics.GetVar("tid")%>'
           site='<%=ics.GetVar("site")%>' />
        ```

> **Note**
>
> Slots cannot include or refer to content of any kind. They must be created as empty placeholders. For rules on designing slots, see "Designing Slots," on page 478.

- Format the slots as you require. For example, if you created slots in the context of a table, you can specify borders directly in the template code or in a style sheet.

For an example of how the `<insitetemplating:slot>` tag is used in context, see page 476.

2. When you finish creating the master template, test it by following the steps on page 482.
3. The master template is like any other JSP template, except for the slots it contains. The template can be shared and edited. For instructions, see "Managing Template, CSElement, and SiteEntry Assets," on page 471.

Testing Master Templates

Testing a master template involves creating a publishable page from the template.

Pre-requisites

Before using a master template, make sure the following requirements are satisfied:

- A master template has been created as explained in "Creating a Master Template," on page 481.
- If you need to create multiple pages from a master template, create a uniquely named copy of the master template for each page that you require. You will then modify the copies.
- If you plan to assign pagelets to slots:
  - The pagelets must exist (they cannot be created from within the InSite Templating feature).
  - The pagelets must not contain slots.
  - The pagelets must be assigned to a category whose name begins with `IT_` (uppercase characters).
  - Each pagelet must be coded to render the required asset(s). A pagelet can call a template and pass the parameters that are needed for rendering.
- If you plan to assign asset-template pairs to slots:
  - The assets and templates must exist (they cannot be created from within the InSite Templating feature).
  - The templates must not contain slots.
- Use Internet Explorer 5.5 or 6.0.

Procedures

Using the steps below, you will create a web page from a master template.

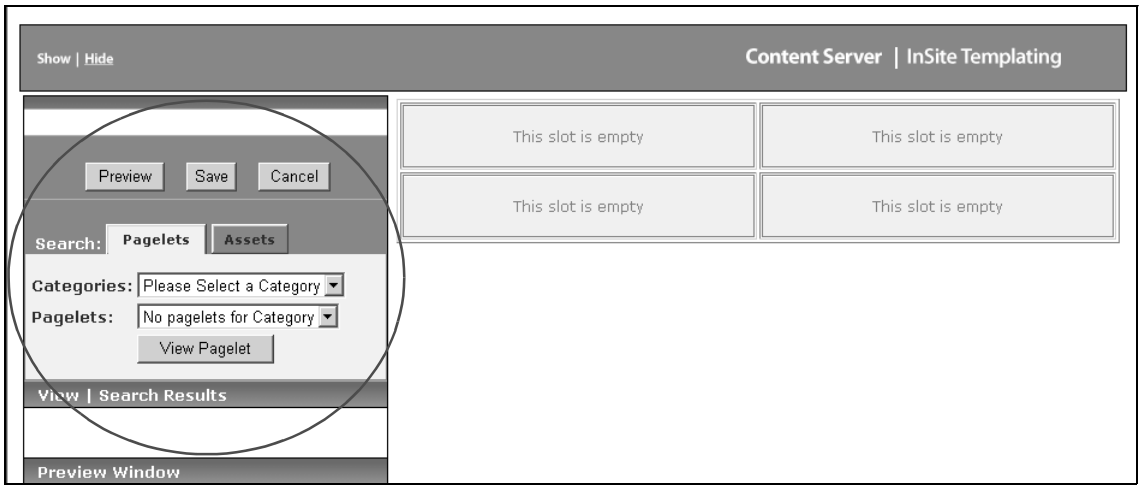
To create a publishable page

1. Make sure the pre-requisites above are satisfied.
2. Locate the master template:

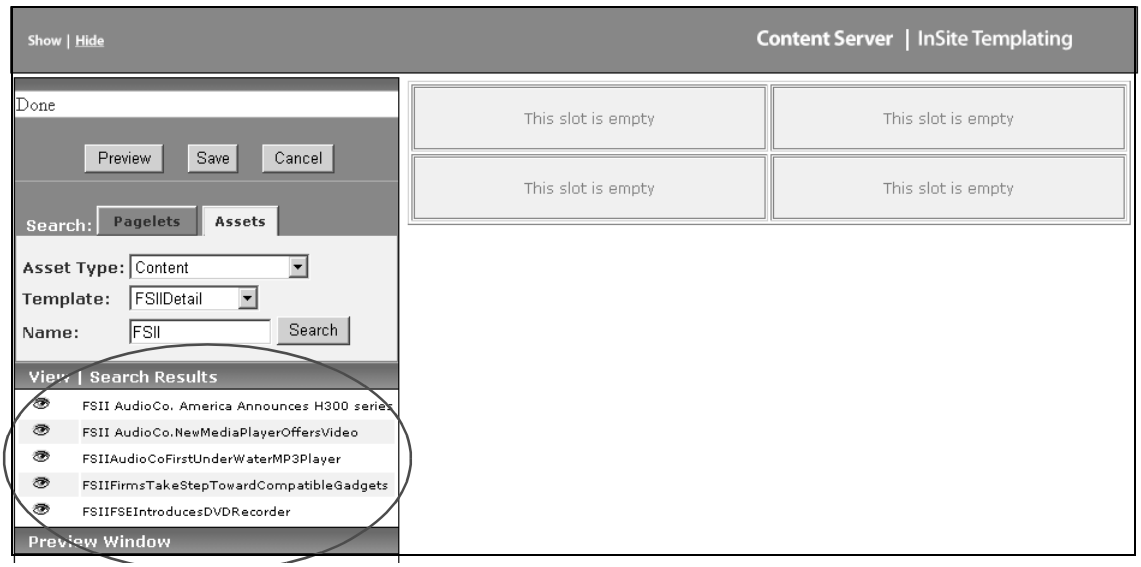
- If you created a typed template, search for an asset of the type that is specified for the template. (For example, if the template is for asset type “Content,” search for an asset of type “Content.”) From the list of results, choose an asset and click its preview icon.
- If you created a typeless template, you can preview the master page from any interface that supports previews (for example, placed and unplaced pages in the **Site Plan** tab, or an asset’s “Inspect” screen). You can also perform a search on assets of any type, choose an asset from the results list, and click its preview icon.

a. In the preview screen, go to the “Template” menu, select a master template, and click **Edit Template**.

3. The master template is displayed as an editable master page. The InSite Templating control panel (shown circled) can be toggled on and off. Make sure it is displayed (use the **Show|Hide** toggle in the upper left-hand corner).



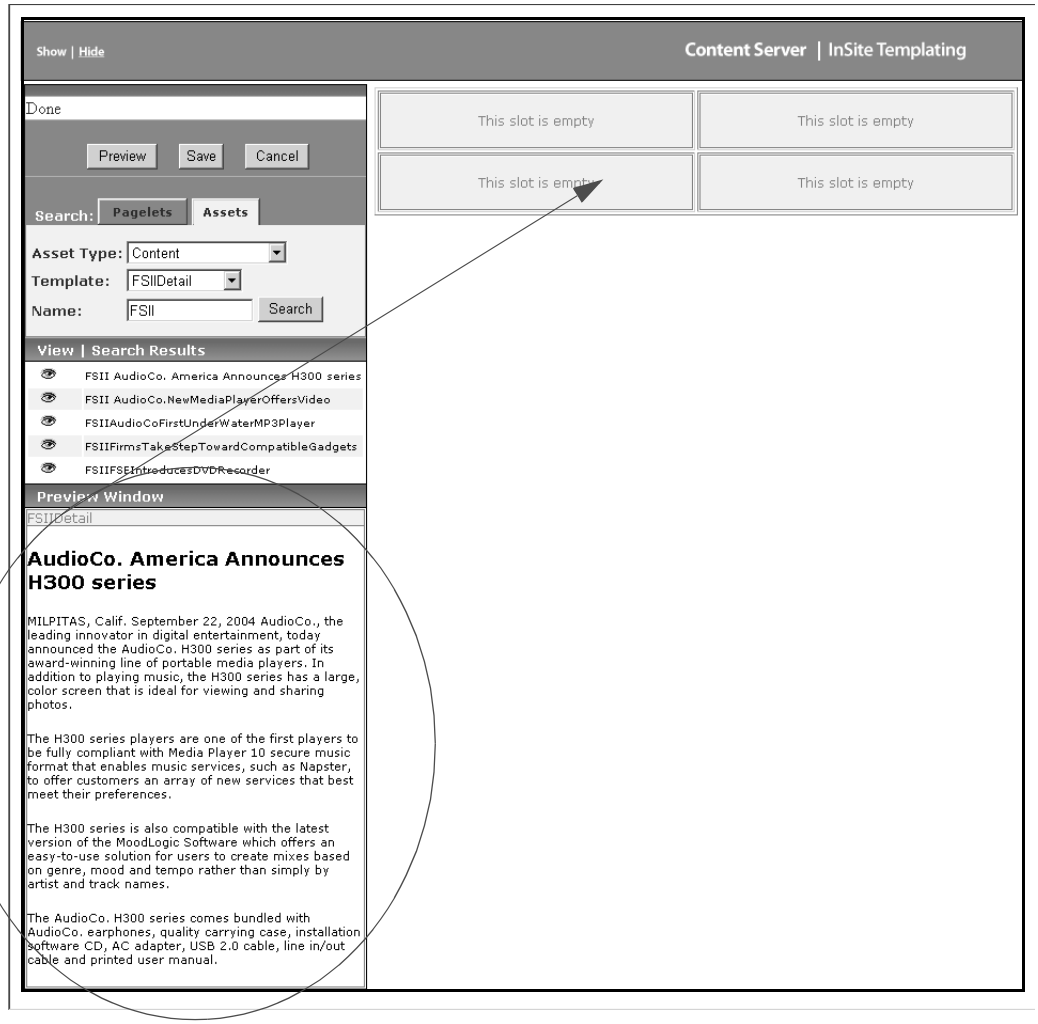
4. Using the control panel, select a preview object (content) for the slots of the master page. The preview object can be a pagelet or an asset-template pair:
   - **To select a pagelet:**
     a) Click the **Pagelets** tab.
     b) Select a category.
     c) Select a pagelet. Make sure the pagelet does not contain slots.
     d) Click **View**.
     e) The pagelet is displayed as a preview object, which you can drag and drop into a slot (go to step 5 on page 485).
   - **To select an asset-template pair:**
     a) Click the **Assets** tab.
     b) Specify the following information:
        - An **asset type**.
        - A **template** that will render assets of the specified type in the format you require. Make sure the template does not contain slots.
        - A **name or value** that will return the assets you are searching for.
     c) Click **Search**.
     d) Select a search result (by clicking its preview icon) to display it as a preview object.



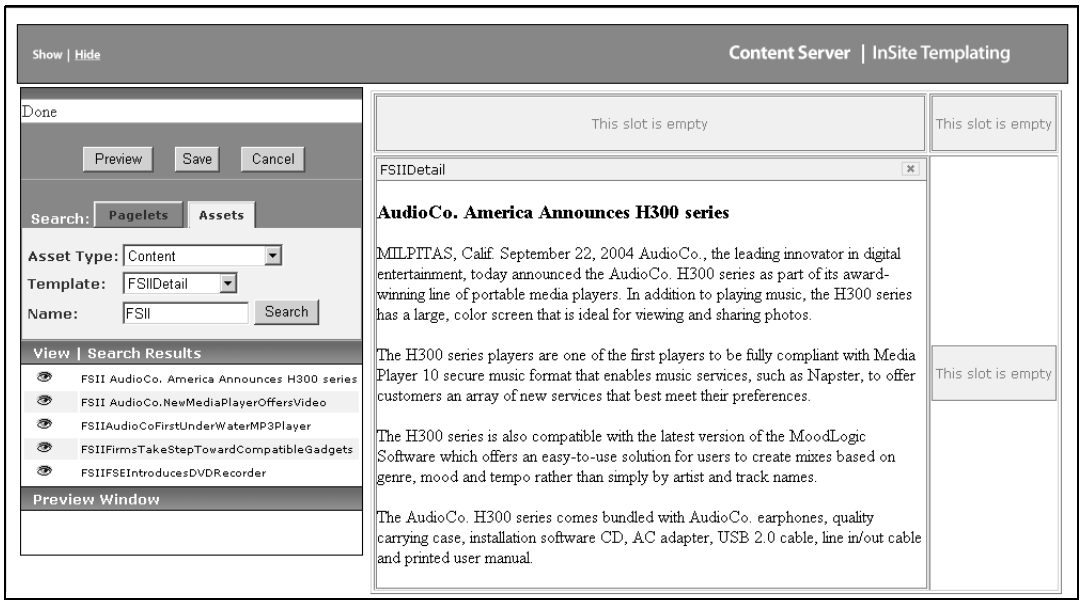
5. Click the preview object **in its upper left-hand corner** and drag it into a slot.

Notes

- If the preview object is clicked anywhere but the upper left-hand corner, the object will not enter the slot. The slot must also be outlined by a dashed line in order for the object to enter.
- Once an object fills a slot, it cannot be returned to the “Preview Window.”



The result of dragging a preview object into a slot is shown below.



6. You can place other preview objects into slots by repeating steps 4 and 5. On pages with multiple slots, at least one slot must be filled. Which slots to fill depends on the layout you require.
7. You can rearrange objects that occupy slots:
   - To reposition an object, drag it (by the upper-left corner) to an empty slot.
   - To interchange the content of two slots, drag one piece of content (by the upper left-hand corner) over the other.
   - To delete an object, click its **x** icon (upper right-hand corner).

> **Note**
>
> Content outside a slot is pre-existing content. It is fixed on the page. It cannot be repositioned or removed except by recoding the master template.

8. Click **Save**.

   The master page is updated. The updates are recorded in the `Template_Composition` table as new parameters for the slots that have somehow been modified (filled with new content or emptied). The original code of the master template remains unaffected.
9. To view the master page as it would appear when published, go back to step 1 on page 482.

Chapter 22

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 Guide*. 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

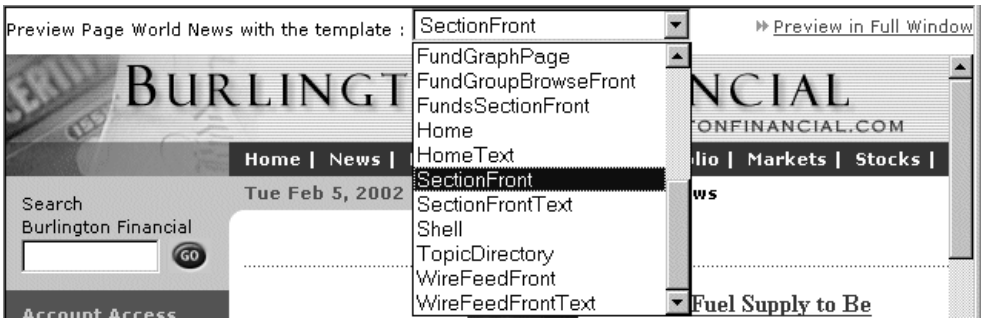
Previewing Assets

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.

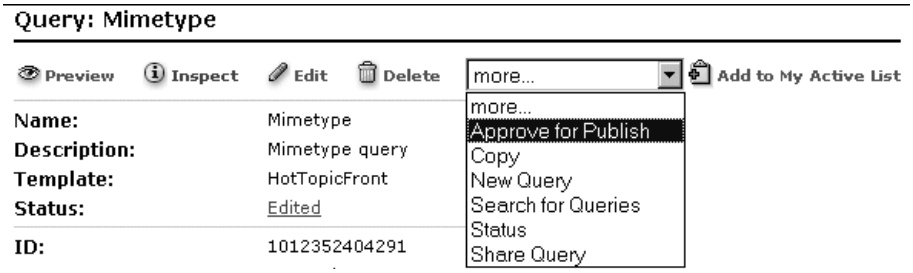
Approving Assets

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 Guide*.

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 10: (Optional) Configure Association Fields" on page 301.

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 User's Guide*.

Creating Collection Assets

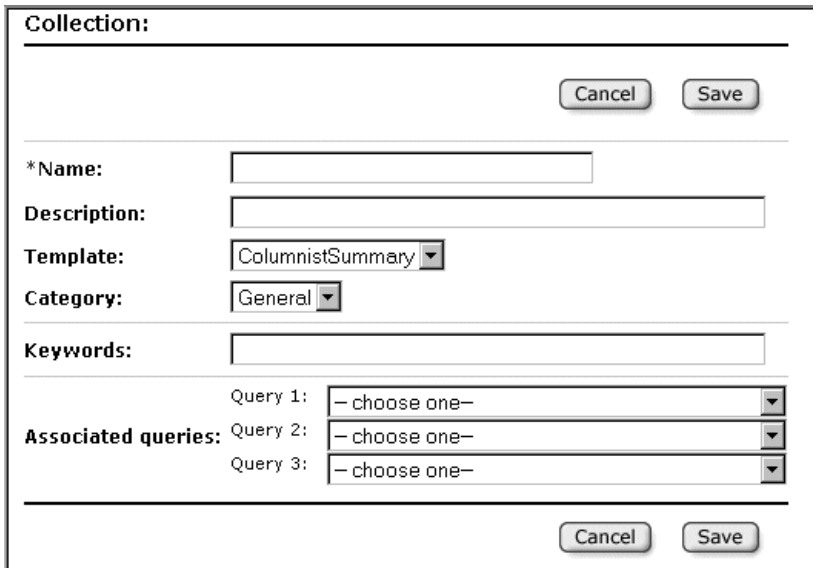
To create a collection asset

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 490.
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 unique, 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.

   The name must be unique for collection assets you are creating for this site.
6. In the **Description** field, type a brief description of the collection. You can enter up to 128 characters.

7. In the **Template** field, select a Template asset from the drop-down list.
8. 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.)
9. 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 User's Guide*.
10. 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.
11. 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 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 the Verity 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

Status Code	Definition
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 12: (Optional) Configure Sources" on page 304.)

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 11: (Optional) Configure Categories" on page 303.)

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.

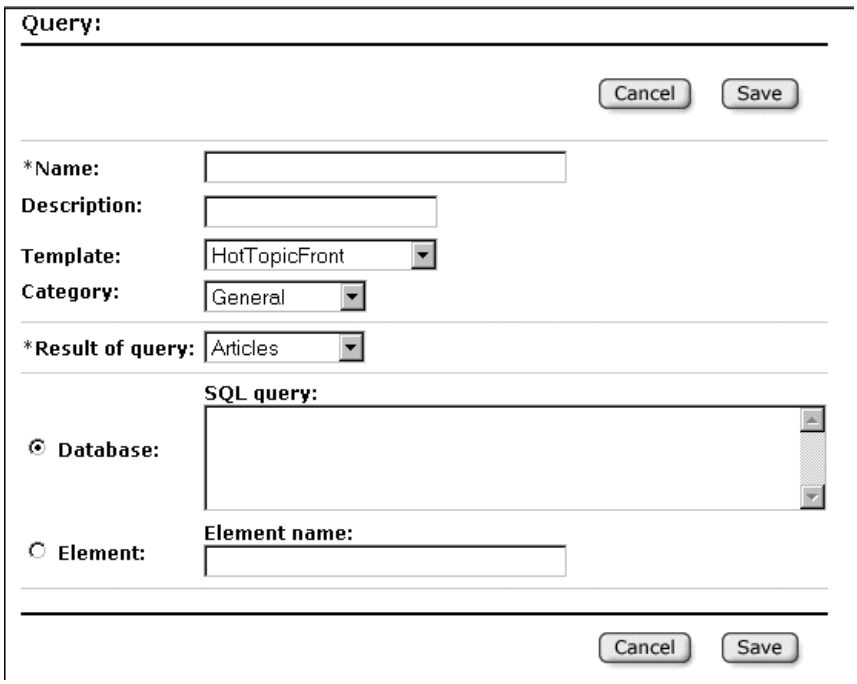
  For example, the **Name** and **Description** fields are required fields for a Burlington Financial article. (The **Description** field is renamed and displayed as **Headline** in the form.) Therefore, the queries for Burlington Financial collections that hold Burlington Financial articles select the **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 unique, 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 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

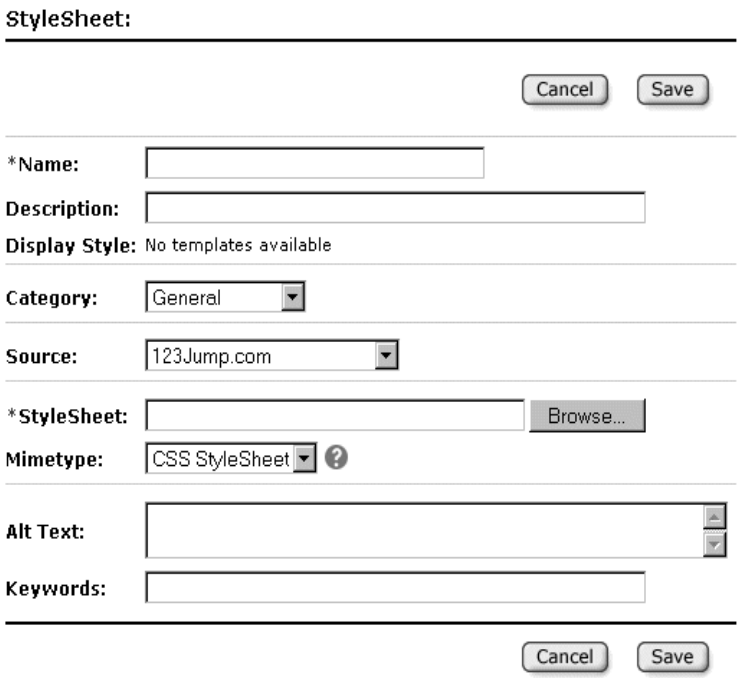
Stylesheet assets store style sheet files of any format (CSS, XSL, and so on). This asset type is installed only if you install the Burlington Financial sample site. Typically, you do not assign Template assets to stylesheet assets because they are, effectively, templates in themselves. However, if you 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 13: (Conditional) Add Mimetypes" on page 305.
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 User's Guide*.
12. Click **Save**.

    CS-Direct uploads the file you specified in the **Stylesheet** field.

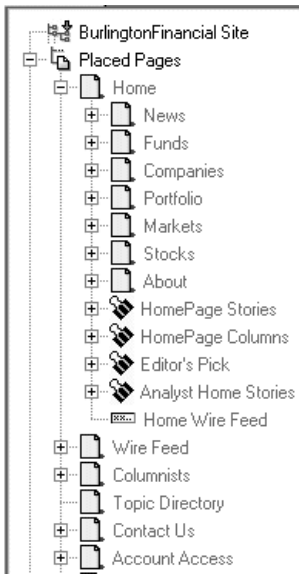
Sharing Stylesheet Assets

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.

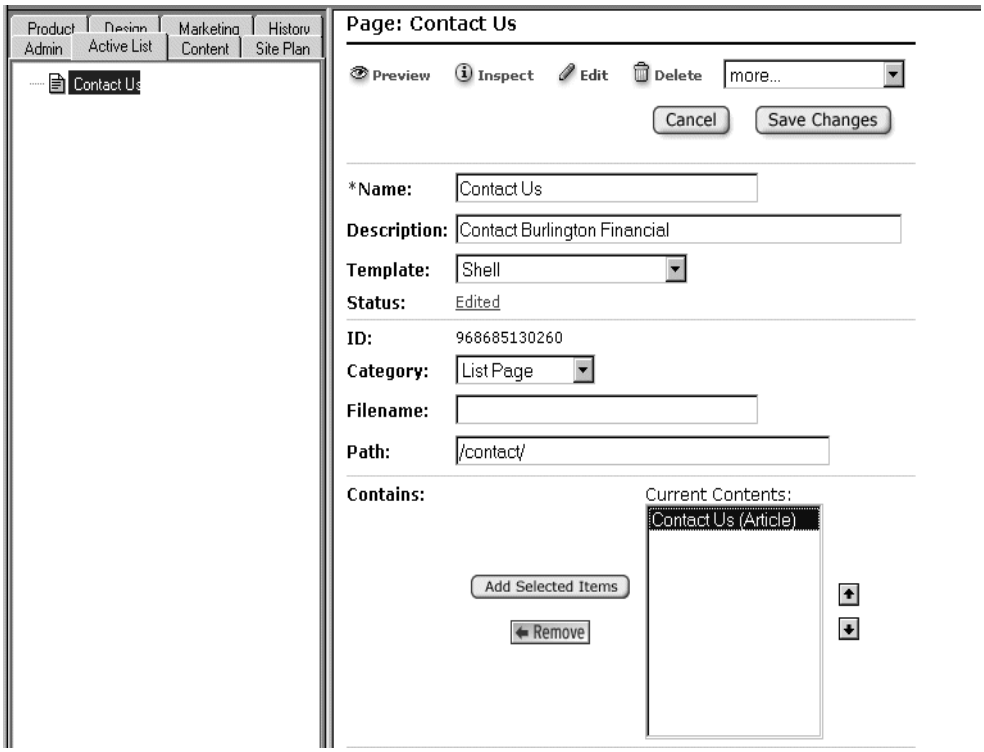
Creating a Page Asset

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 unique, 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.

   The name must be unique for page assets you are creating for this site.
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 191.

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 User's Guide* for instructions about 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 490 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.

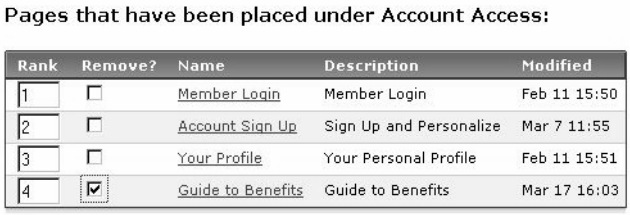
Changing Parent Pages

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:



   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 23

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 20, "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 that 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. The CacheManager servlet 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.

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, for the Export to Disk publishing method, **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 stamp 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.

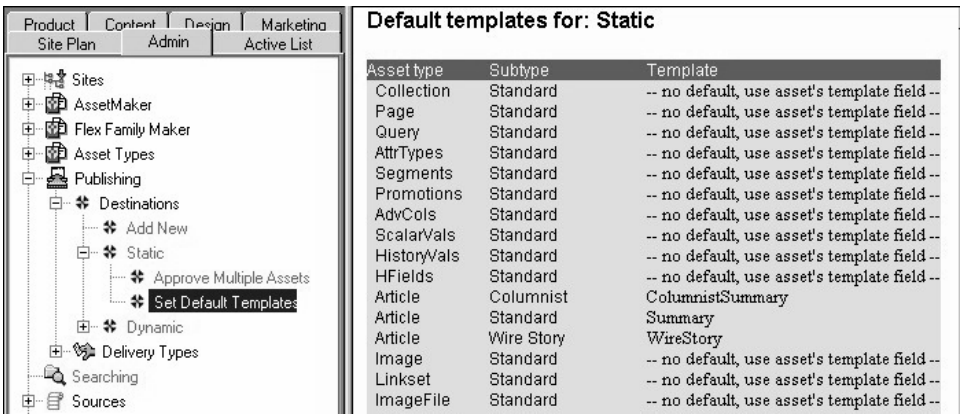
However, when Export to Disk actually publishes the asset, it does not necessarily use the template that is assigned to the asset. Why? Because the code in another element could determine that a different template is used for that asset in certain cases.

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:



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 14, "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 cachekey 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 Mirror to Server 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 509 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 527 and "Searchstates" on page 528.

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 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. This means that you must use this tag carefully because the more pages there are 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), 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, note that the chain of called elements should not be more than 20 levels deep: otherwise the assets that are displayed may behave unpredictably.

Coding Elements that Display Basic Assets

To code an element for a template, you need to understand how the asset type it formats is designed. 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 change.

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 24, "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 content 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 -->

<CSVAR NAME="Variables.asset:description"/><br/>

<!-- 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 541.

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 29, "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 553.

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 541.

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"><csvar
            NAME="ArticlesFromTheQuery.subheadline"/></A><P/>
```

For another example, see "Example 4: Coding Templates for Query Assets" on page 559.

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>
   <B><FONT FACE="Verdana, Arial, Helvetica, sans-serif"><A
   HREF="Variables.referURL" REPLACEALL="Variables.referURL">From
   the Wires...</A>
   </FONT></B>
   </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
```

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 514.

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 10, "Data Design: The Asset Models" and Chapter 15, "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: <CSVAR NAME="pricelist.value"/><P/>
```

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">
         <CSVAR NAME="colorlist.value"/> 
         </LOOP><P/>
```

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">
      <CSVAR NAME="lolist.attributename is "/>
      <LOOP LIST="multi:lolist.attributename">
         <CSVAR NAME="multi:lolist.attributename.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 10, "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.READDATA ID="descFile.value" LISTVARNAME="descData"/>
<CSVAR NAME="descData.@urldata"/>
<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="descData.@urldata"
   BLOBKEY="Variables.idColumn"
   BLOBCOL="Variables.uColumn"
   BLOBHEADERNAME1="application/pdf"
   BLOBHEADERVALUE1="a CLASS='thumbhead' href"/>
   <CSVAR NAME="View Description (PDF)"/>  <![CDATA[</a>]]>
```

The RENDER.SATELLITEBLOB tag returns a BlobServer URL in an HREF tag (SERVICE="a CLASS='thumbhead' href") and the CSVAR tag displays the link.

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"/>
<CSVAR NAME="descData.@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?
<CSVAR NAME="Variables.count"/><P/>
```

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">
      <CSVAR NAME="colors.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">
      <CSVAR NAME="distinct:color.value"/> 
</LOOP><P/>

Here are all the possible styles:
<LOOP LIST="distinct:style">
      <CSVAR NAME="distinct:style.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?
<CSVAR NAME="Variables.count"/><P/>
```

And 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"?
<CSVAR NAME="Variables.count"/><P/>
```

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?
<CSVAR NAME="Variables.count"/><P/>
```

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 534.

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 534.

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 you are debugging your code, don't forget that you can use the Page Debugger utility. There are also additional debugging properties on the **Debug** tab in the `futuretense.ini` file that you can enable, if necessary. When you enable these properties, additional error and debugging messages are then written to the `futuretense.txt` log file, which is located in the Content Server installation directory.

For information about the debugging properties, see the *Content Server Property Files Reference*. For information about the Page Debugger tool, see Chapter 7, "Content Server Tools and Utilities."

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 <CSVAR NAME=“Variables.errno”/>
            while rendering <CSVAR NAME=“pagename”/>
            with asset ID <CSVAR NAME =“Variables.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 544, 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 Mirror to Server Publish Sessions” on page 512, 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 24

Template Element Examples for Basic Assets

This chapter uses examples from the Burlington Financial sample site to illustrate the information presented in Chapter 23, “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.

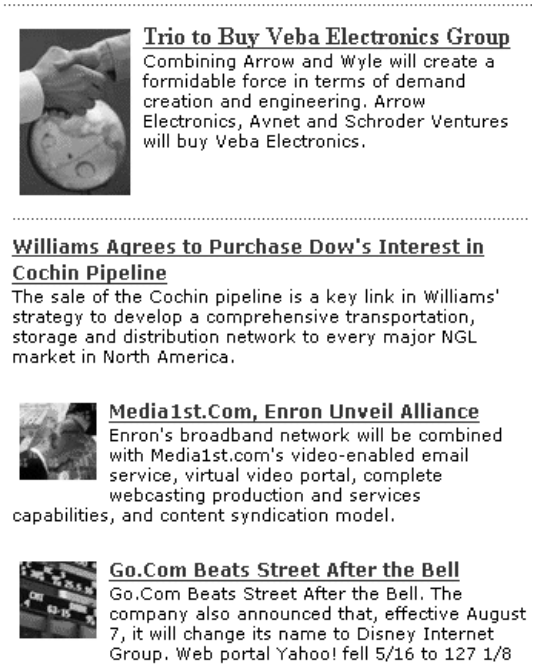
Example 1: Basic Modular Design

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`:

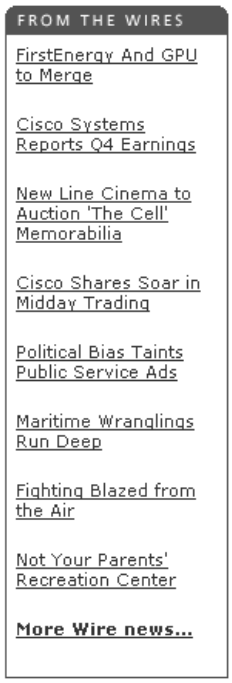
```
<CALLELEMENT NAME="BurlingtonFinancial/Common/TextOnlyLink">
   <ARGUMENT NAME="ct" VALUE="ColumnistFrontText"/>
   <ARGUMENT NAME="assettype" VALUE="Page"/>
</CALLELEMENT>
```

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:

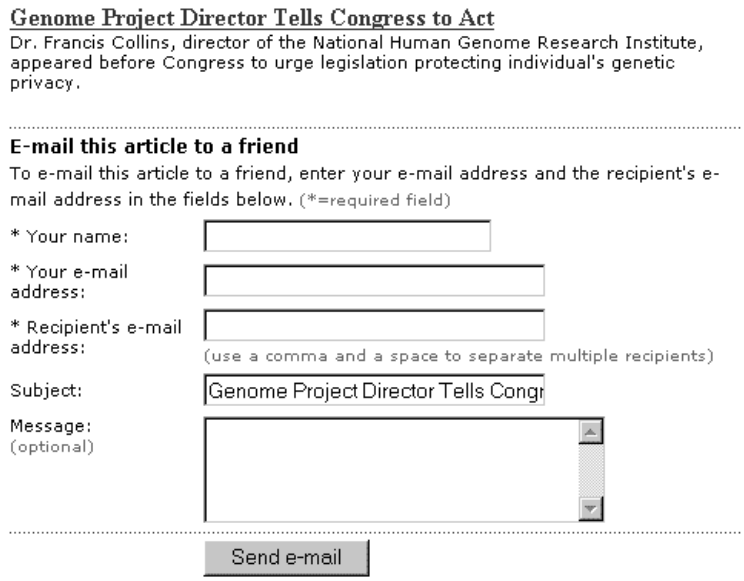
```
<RENDER.FILTER LIST="ArticlesFromWireQuery"
LISTVARNAME="ArticlesFromWireQuery" LISTIDCOL="id"/>
   <if COND="ArticlesFromWireQuery.#numRows!=0">
   <then>
   <LOOP LIST="ArticlesFromWireQuery">
   <RENDER.GETPAGEURL PAGENAME="BurlingtonFinancial/Article/
   WireStory"
   cid="ArticlesFromWireQuery.id"
   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 548 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 rootelement. 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 104.)

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 25

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 dir:addattrs	Adds attributes to an existing entry (which can be either a user or a group).
DIR.ADDGROUPMEMBER dir:addgroupmember	Adds a member to a group (usually a user).
DIR.CHILDREN dir:children	Retrieves the child entries for a specified parent in a list variable.
DIR.CREATE dir:create	Creates a directory entry.
DIR.DELETE dir:delete	Deletes a directory entry.
DIR.GETATTRS dir:getattrs	Gets the attribute values for a specified entry in a list variable.
DIR.GROUPMEMBERS dir:groupmembers	Lists the members of a specified group.
DIR.GROUPMEMBERSHIPS dir:groupmemberships	Lists all the groups that an entry (either a group or a user) belongs to.
DIR.LISTUSERS dir:listusers	Returns a list of all the users in the directory.
DIR.REMOVEATTRS dir:removeattrs	Deletes an attribute value for an entry.
DIR.REMOVEGROUPMEMBER dir:removegroupmember	Removes an entry from a group.
DIR.REPLACEATTRS dir.replaceattrs	Replaces the value of an attribute for an entry (either a user or a group).
DIR.SEARCH 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 572.

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 9, “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` `acl:create`	Creates an ACL
`ACL.DELETE` `acl:delete`	Deletes an ACL
`ACL.GATHER` `acl:delete`	Gathers fields into an ACL
`ACL.GET` `acl:get`	Copies a field from an ACL
`ACL.LIST` `acl:list`	Retrieves a list of ACLs
`ACL.LOAD` `acl:load`	Loads an ACL
`ACL.SAVE` `acl:save`	Saves an ACL
`ACL.SCATTER` `acl:scatter`	Scatters a field from an ACL
`ACL.SET` `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 user:login	Logs a user in.
USER.LOGOUT user:logout	Logs a user out.
USER.SU 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 26

The HelloAssetWorld Sample Site

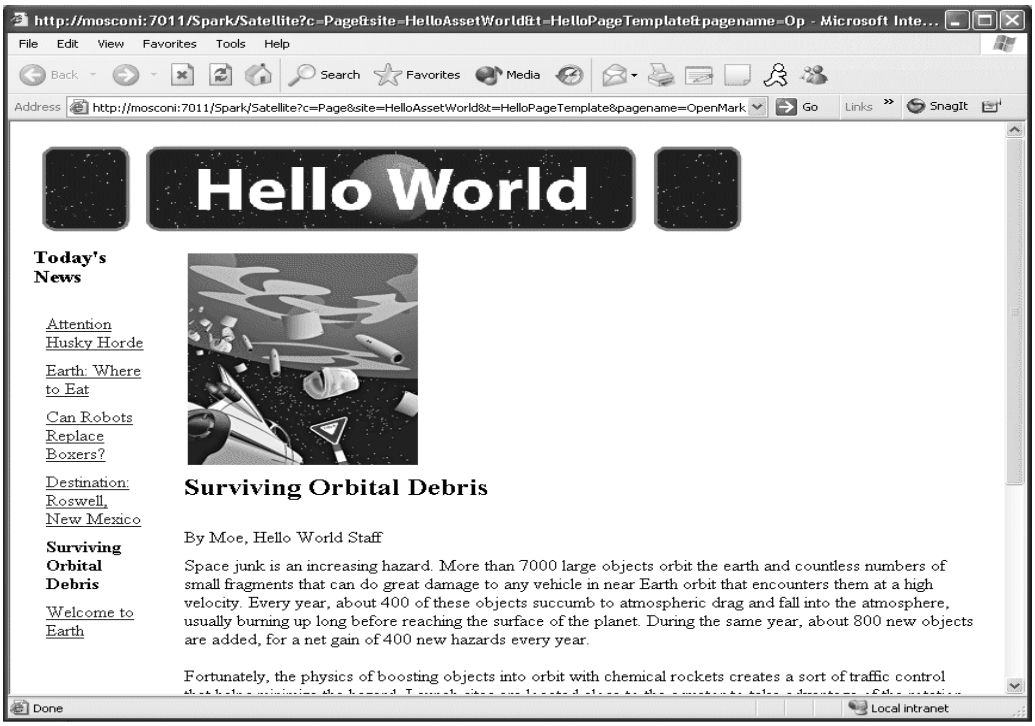
The HelloAssetWorld sample site is a sample web site built using Content Server and CS-Direct. It is meant to provide a simple entry point into the process of building a web site with CS-Direct. This chapter focuses on the steps that a developer would take in creating this simple web site; further information on HelloAssetWorld's configuration and users are in the *Content Server Administrator's Guide* and the *Content Server User'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 27, "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 contianing 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 27

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:

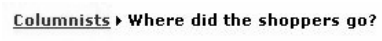
Because CS-Direct allows assets to be assigned to multiple parents at the same time, the same article can appear as a member of a collection on the Home page and on the News page at the same time. There is no way to identify the true parent of the article, so Burlington Financial passes the `id` of the 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:



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, for example the Verity search engine.

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.

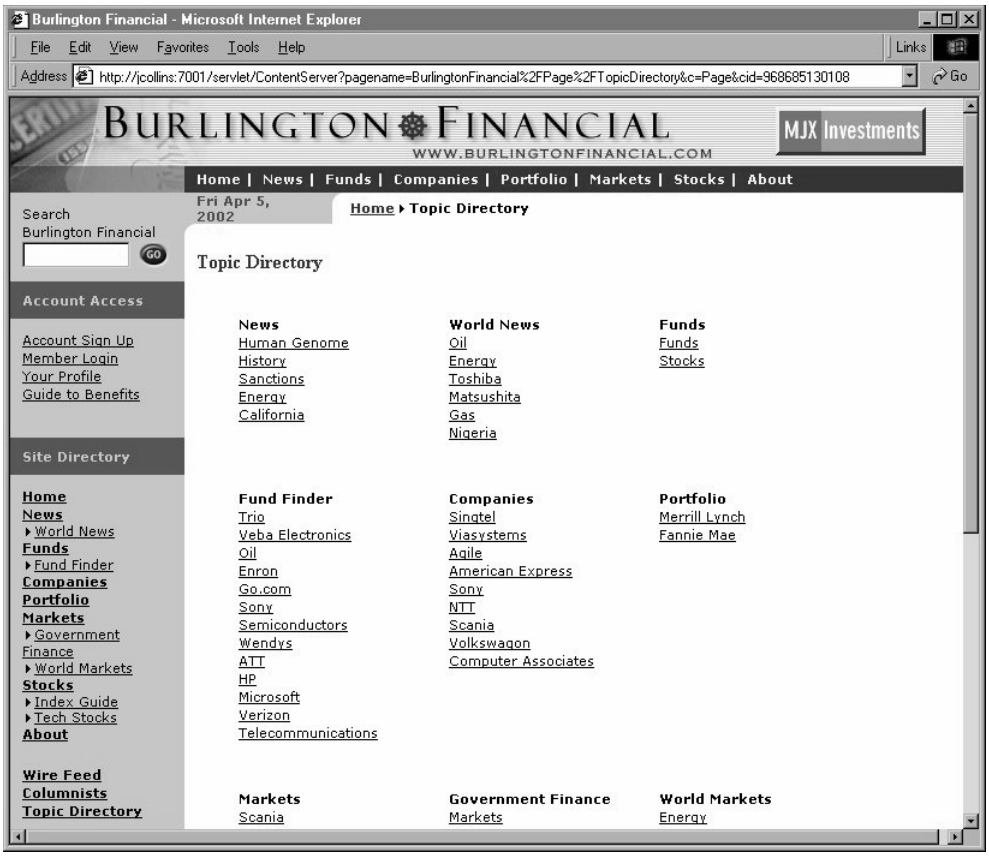
When visitors click a link, they are taken to a page that renders the query asset HotTopics, using the HotTopicFront template. The HotTopics query does a straight SQL match against articles that contain the selected keyword. The keyword search is not constrained in any way—it searches all articles in 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 25, “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

```
1  <SEARCHSTATE.CREATE NAME="ss"/>
2  <ASSETSET.SETSEARCHEDASSETS NAME="as"
   ASSETTYPES="ProductGroups" CONSTRAINT="ss"/>
3  <ASSETSET.GETATTRIBUTEVALUES NAME="as"
   TYPENAME="PAttributes" ATTRIBUTE="FundFamily"
   LISTVARNAME="fflist"/>
4
5  <P><SELECT name="FundFamily" SIZE="3" MULTIPLE="1">
6  <OPTION SELECTED="" VALUE="NoPreference"/>No Preference
7  <LOOP LIST="fflist">
8  <OPTION/><csvar NAME="fflist.value"/>
9  </LOOP>
10  </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 ana 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 28, “Customizing the User Interface”
- Chapter 29, “Coding for the InSite Editor”
- Chapter 30, “Customizing Workflow”

Chapter 28

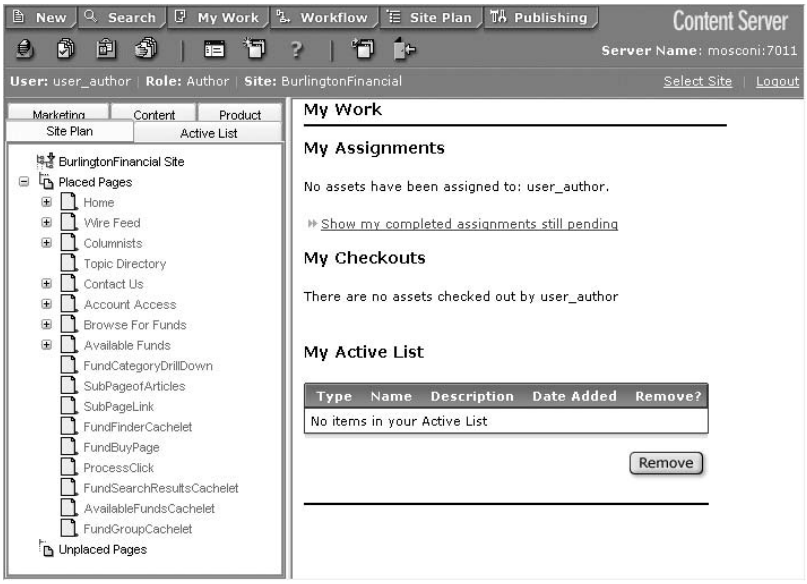
Customizing the User Interface

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
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 Type	Location	Description
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 4: 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.
`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 5: Tree-Specific Parameters

Parameter	Description
`Title`	Sets the text that is displayed on the tab. 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. 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. 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. 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 (+). 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 (-). 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. 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. 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 6: 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. 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. 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.

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. 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: Product Tab > Reebok > Running Shoes the value of LoadURL is as follows: ContentServer?pagename=OpenMarket/Gator/UIFramework/LoadTab&AssetType=ProductGroups&populate=OpenMarket/Xcelerate/AssetType/ProductGroups/LoadTree&op=load&parent=Variables.parentid 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. 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 29

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:

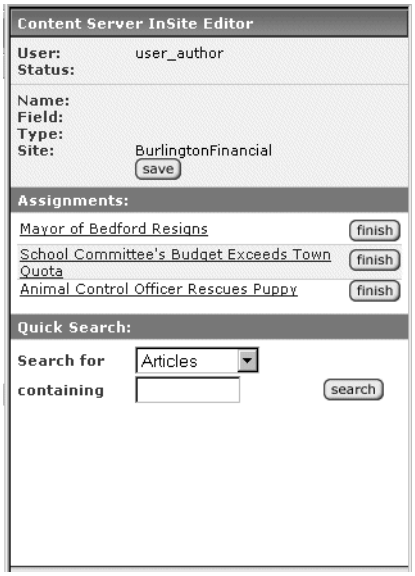
- Setting the `xcelerate.enableinsite` property in the `futuretense_xcel.ini` file to `true`.
- Coding templates that invoke the InSite Editor feature for the fields that you want content providers to be able to edit in this way.

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

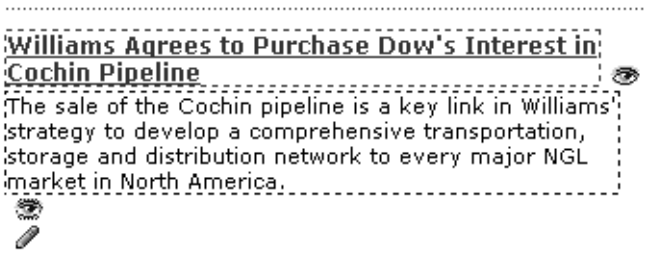
When a Content Server user with InSite Editor privileges previews any asset on a management system that has the InSite Editor configured, a separate control panel window appears, similar to the following:



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 30

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 asignee -->
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 31, “Overview of Web Services”
- Chapter 32, “Creating and Consuming Web Services”

Chapter 31

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.

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 5.5.1 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`

For complete information about the WSDL files supplied with Content Server, refer to the *Content Server Web Services Reference*.

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 32

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
- Consuming 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

Note

For complete information about WSDL files, supported operations, and required inputs, refer to the *Content Server Web Services Reference* manual.

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

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

Data for the predefined WSDL operations is passed using RPC-style interactions (versus exchanging entire XML documents) to your program. Data types for all possible inputs for the predefined web services are described in the *Content Server Web Services Reference* manual. 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. These functions, including Java example code, are described in the *Content Server Web Services Reference* manual.

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
2. Creating a Content Server Page
3. Writing a Content Server Element
4. Creating a WSDL File

Process Flow

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

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. Enter the location of the element for your client function in the `SiteCatalog` table.
2. Start Content Server Explorer and log in to Content Server. For instructions, refer to the online help or the instructions in this guide.
3. In the left pane, open the `SiteCatalog` table.
4. Select the folder for your site.
5. In the **pagename** field, create an entry for the last part of the page name for your web service in the `SiteCatalog` table.
6. 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.)
7. 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.
8. In the **csstatus** field, enter `Live`, or the appropriate status at this time.
9. 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. Keep in mind that although comments are removed when the `XMLdebug` property is turned off, extra whitespace can still corrupt the XML stream. 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 5.5.1 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>
```

Consuming Web Services

Content Server can interact with any remote application that offers a web service and that returns a data type supported by the XML Schema.

Using the information contained in the WSDL file for a given web service, you use a supplied Content Server invocation tag to specify the location of the web service, the operation to invoke, and the name of the object in which the return data is to be stored. An associated parameter tag specifies the input parameters for the particular operation.

Once the data is transmitted and stored in Content Server, it becomes available for display or further processing. Content Server pages or APIs handle the return data according to your custom business logic. By configuring the web service as a tag, Content Server handles data as if it were an ordinary content tag for a native Content Server function.

If you are not using a tool that automatically creates the client, you need to read the WSDL file and write your code by hand. Note that for automated client generation, the output language of the client depends on the tool you select.

Locating the Web Service

The web service description should be in WSDL format. If you are not given the location of the WSDL, you may need to search for the web service via UDDI in a web services directory.

Gathering Information from the Remote WSDL File

Enter the URL of the WSDL file in a browser and view its contents. The WSDL file defines a service and port for each of one or more operations. Depending on the WSDL, you may see multiple operations that correspond to multiple servers, or multiple operations that correspond to a single server.

In the WSDL, look for the following XML elements and record their values:

- 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.

Providing Information to Content Server

Information about the remote web service that is contained in the WSDL file is transmitted to Content via SOAP tags, which you configure.

To create a Content Server page and element that includes SOAP tags

1. Create a Content Server page and element.

2. In the element, include the following SOAP tags:
   a. For the `webservices.invoke` tag, set parameters for the target namespace, service name, and port name for the web service. Also set the `object` parameter, which specifies the name of the object that will contain the return data.
   b. For the `webservices.parameter tag`, set parameters that represent inputs for the web service operation.

SOAP Tag Example

The following element includes the Content Server SOAP tags for consuming web services:

```
<webservices:invoke
    wsdl="http://soapinterop.java.sun.com/round2/base?WSDL"
    target="http://soapinterop.org/"
    service="Round2Base"
    port="RIBaseIFPort"
    operation="echoString"
    object="echostringjsp">
   <webservices:parameter type="string" value="hello world jsp"/>
</webservices:invoke>
<%=ics.GetObj("echostringjsp")%>
```

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 33, “Creating Visitor Data Assets”
- Chapter 34, “Recommendation Assets”
- Chapter 35, “Coding Engage Pages”

Chapter 33

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 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.

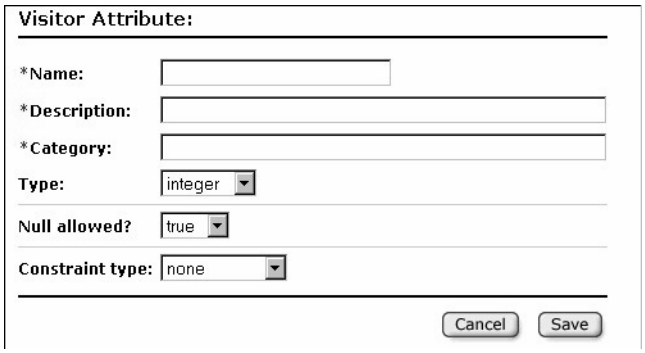
Creating Visitor Attributes

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 **ScalarVals** 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 brief description of the attribute (field). You can enter up to 128 alphanumeric characters but you should keep this description as short as you can because **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 forms. 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.
1. 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.
2. 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.
3. 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.

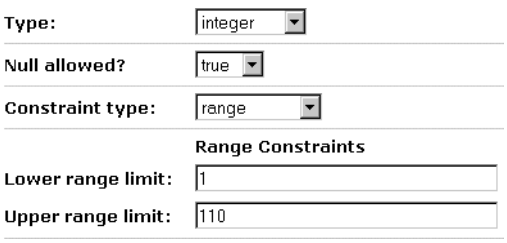
1. In the **Constraint type** field and select range.

   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

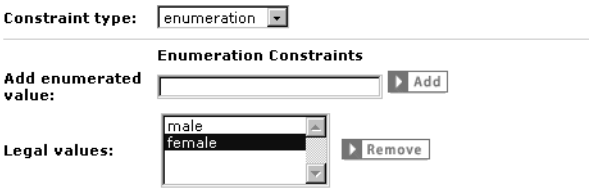
1. In the **Constraint type field** and select enumeration.

   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 **Other Publications** list, select the name of a site and click the arrow button to move it to the Selected list.
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.

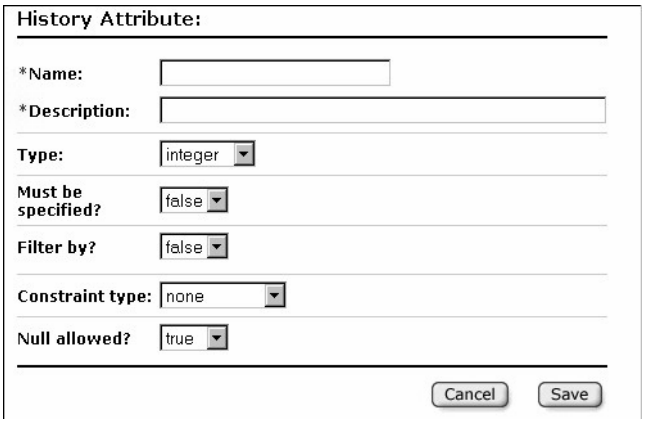
Note

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 brief description of the attribute (field). You can enter up to 128 alphanumeric characters but you should keep this description as short as you can because **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.

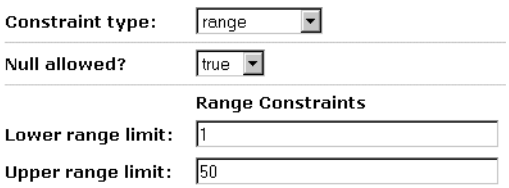
Option 2: Configure the attribute to accept input from a range of values

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.)

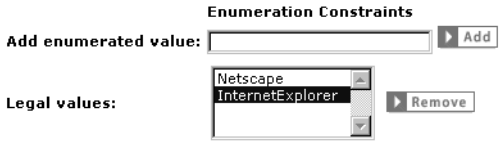
   For example, an attribute named "Number of Items" can be restricted to accept only values between 1 and 50.



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.

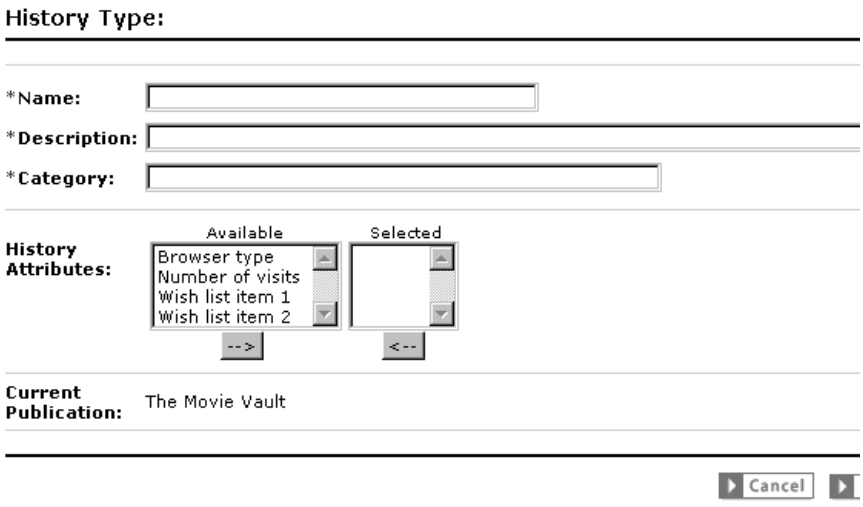
Creating History Definitions

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 brief description of the history definition (record). You can enter up to 128 alphanumeric characters but you should keep this description as short as you can because **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 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 User's Guide*.

Chapter 34

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 35, "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 35, "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 User’s Guide*.

Chapter 35

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 a 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.

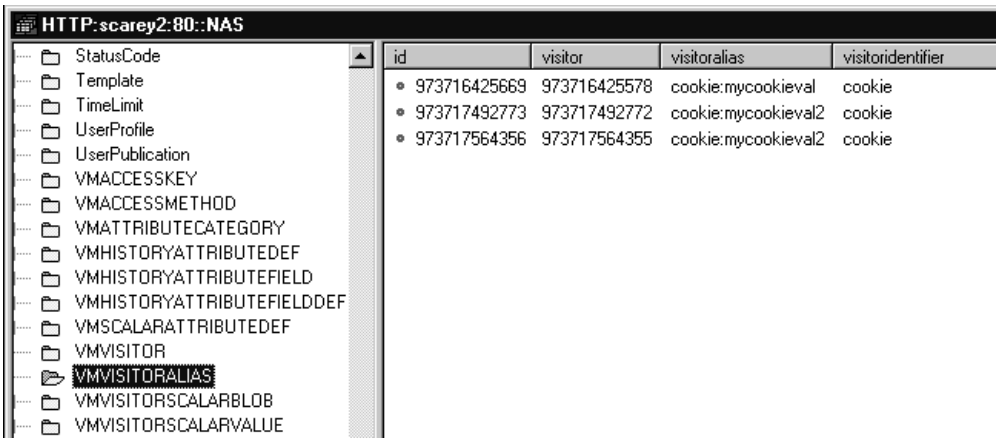
Identifying Visitors and Linking Sessions

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` `vdm:setscalar`	`VMVISITORSCALARVALUE`
`VDM.SAVESCALAROBJECT` `vdm:savescalarobject`	`VMVISITORSCALARBLOB`
`VDM.RECORDHISTORY` `vdm:recordhistory`	`VMz----------` (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 9, "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 33, "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 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, or by examining it with the Page Debugger utility.

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.

Appendixes

This part contains the following appendixes:

- Appendix A, “Creating a Hierarchical Flex Family”
- Appendix B, “Content Server URL Assemblers”
- Appendix C, “White Space and Compression”

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 attributes and single-valued definitions. 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

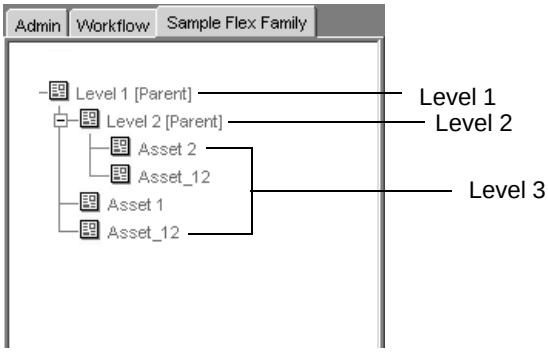
This section describes the flex family and hierarchical structure that you will be creating.

Hierarchical Organization

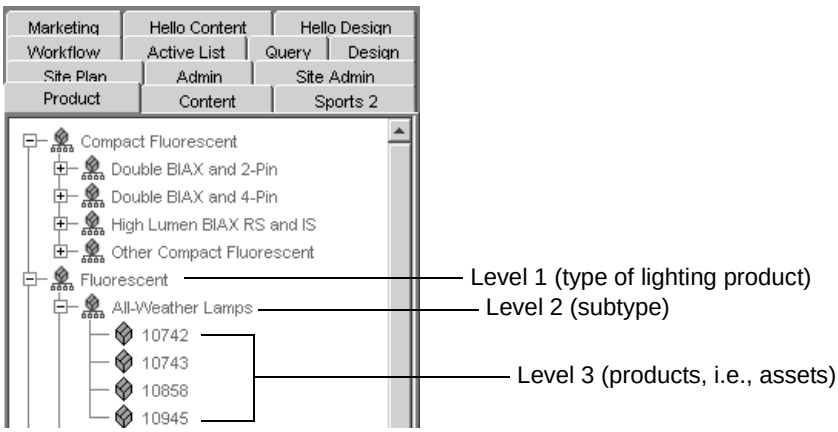
The flex family that you will be creating in this tutorial consists of three levels:

- A top-level parent named Level 1 [Parent]
- A second-level parent named Level 2 [Parent]
- 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 below:



An example of such a hierarchy can be found on the GE Lighting sample site (on the "Product" tree tab, shown below).



The rest of this section provides specifications for the flex family and its required members, which you will create. Note that flex filters are optional components; they are not included in this tutorial.

Flex Family Specifications

Table 7 lists the flex family members that you will be creating in this tutorial.

Table 7: Flex Family Members

Member	Name
Flex Attribute Type	SampleAttribute
Flex Parent Definition Type	SampleParentDefinition
Flex Definition Type	SampleAssetDefinition
Flex Parent Type	SampleParent
Flex Asset Type	SampleAsset
Flex Filter Type	—

Table 8 lists specifications for the flex family members that you will be creating.

Table 8: Specifications for Flex Family Members

Family Member	Instances		Based on Parent Definition	Based on Asset Definition
SampleAttribute (Type)	Sample Attributes:	Attribute 1 Attribute 2		
SampleParent Definition (Type)	Sample Parent Definitions:	Level 1	—	
		Level 2	Level 1	
SampleAsset Definition (Type)	Sample Asset Definitions:	Level 1 Asset Def	Level 1	
		Level 2 Asset Def	Level 2	
		Level_12 Asset Def	Level 1 *and* Level 2	
SampleParent (Type)	Sample Parents:	Level 1 [Parent]	Level 1	
		Level 2 [Parent]	Level 2	
SampleAsset (Type)	Sample Assets:	Level 1 Asset Level 2 Asset Level_12 Asset		Level 1 Asset Def Level 2 Asset Def Level_12 Asset Def

Procedures

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. On the **Admin** tab, expand **Flex Family Maker** and double-click **Add New Family**.
4. In the "Flex Family Maker" form, fill in the fields exactly as shown below:

Field Name	Enter	Comments
Flex Attribute	SampleAttribute	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	SampleParentDefinition	
Flex Definition	SampleAssetDefinition	
Flex Parent	SampleParent	
Flex Asset	SampleAsset	
Flex Filter		

5. Click **Continue**.
6. 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.
   b. Click in the **Plural** field and enter the plural form of the name used in the previous step.
   c. Click **Add New Flex Family**.
7. Wait for Content Server to create the flex family and return the message indicating that the flex family members (asset types) were successfully installed.
8. 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 Hello Asset World 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 Hello Asset World 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**.
   c. From the list, select the asset types that you just created (SampleAsset, SampleAssetDefinition, SampleAttribute, SampleParent, and SampleParentDefinition).
   d. Click **Enable Asset Types**.
   e. 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: Set Up Content Server's Tree

In this step, you will set up Content Server to display a tree tab that tracks the creation of your flex family. The tab will display selected members of the flex family, when you finish creating each member.

To set up Content Server's tree

1. On the **Admin** tab, double-click the **Tree** node.
2. In the "Tree Tab" form, scroll to the bottom and select **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	
Tooltip	Sample Flex Family	
Sites	HelloAssetWorld (or a site of your choice)	
Required Roles	Any	
Tab Contents	Sample Parent Definition Sample Parent Sample Asset Sample Asset Definition **Note:** Use the "Display Order" arrow to arrange the members as shown above.	

4. Click **Save**.
5. Refresh the screen to display the new tree tab.
6. Go to the next step.

Step 4: Create Flex Attributes

In this step, you will create two flex attributes for the flex family.

To create the first flex attribute

1. From the button bar, click **New**.
2. From the "New" list, select **New Sample Attribute**.
3. In the "Sample Attribute" form, fill in the following fields:

Field Name	Enter or Select	Comments
Name	Attribute_1	The name cannot contain spaces.
Description	Attribute 1	
Value type	string	
Number of values	single	

4. Click **Save**.

To create the second attribute

1. From the button bar, click **New**.
2. From the "New" list, select **New Sample Attribute**.
3. In the next form, fill in the following fields:

Field Name	Enter or Select	Comments
Name	Attribute_2	The name cannot contain spaces.
Description	Attribute 2	
Value type	string	
Number of values	single	

4. Click **Save**.
5. Go to the next step.

Step 5: Create Parent Definition Assets

In this step, you will create two parent definition assets, using the attributes you created in the previous step. The first parent definition asset establishes the top level of the hierarchy; the second parent 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 Sample Parent Definition**.
3. In the next form:
   a. Fill in the fields as follows:

Field Name	Enter or Select	Comments
Name	Level 1	
Description	Level 1	
Parent Select Style	Select Boxes	
Sample Parent Definitions		No sample parent definition is selected. Therefore, this current parent definition establishes the first level of the hierarchy.
Attributes	Attribute_1 **Note:** Click the **Required** arrow to move your selection to the "Selected" list box.	This attribute will be passed on to parent assets when you base the parent assets on the current parent definition.

   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 Sample Parent Definition**.

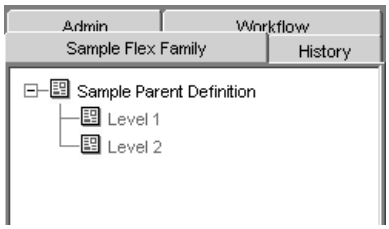
3. In the next form:

   a. Fill in the fields as follows:

Field Name	Enter or Select	Comments
Name	Level 2	This parent definition establishes the second level of the hierarchy.
Description	Level 2	
Parent Select Style	Select Boxes	
Sample Parent Definitions	Level 1 **Note:** Under "Single Value," click the **Required** arrow to move your selection to the "Selected" list box.	Selecting Level 1 subordinates the current parent definition to the Level 1 parent definition. When parents are created and based on the current parent definition, they are subordinated to parents that are based on the Level 1 parent definition.
Attributes	Attribute_2 **Note:** Click the **Required** arrow to move your selection to the "Selected" list box.	This attribute will be passed on to parent assets when you base the parent assets on the current parent definition.

   b. Click **Save**.

4. Go to the "Sample Flex Family" tab and expand its contents to make sure they are identical to the display below:



5. Go to the next step.

Step 6: Create Flex Parent Assets

In this step, you will create flex two 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 Sample Parent**.
3. In the form that appears, fill in the fields as follows:

Field Name	Enter or Select	Comments
Name	Level 1 [Parent]	
Sample Parent Definition	Level 1	Choosing Level 1 places the parent you are creating at the top level of the hierarchy.

4. In the next form:
   a. Click in the "Attribute_1" field and enter **Attribute 1 value**.
   b. Click **Save**.

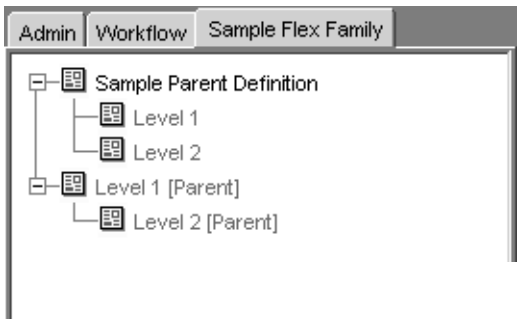
To create the second-level parent of the hierarchy

1. From the button bar, click **New**.
2. From the list of options that appears, select **New Sample Parent**.
3. In the form that appears, fill in the fields as follows:

Field Name	Enter or Select	Comments
Name	Level 2 [Parent]	
Sample Parent Definition	Level 2	Choosing Level 2 places the parent you are creating at the second level of the hierarchy.

4. In the next form:
   a. Click in the "Attribute_2" field and enter **Attribute 2 value**.
   b. Click **Save**.

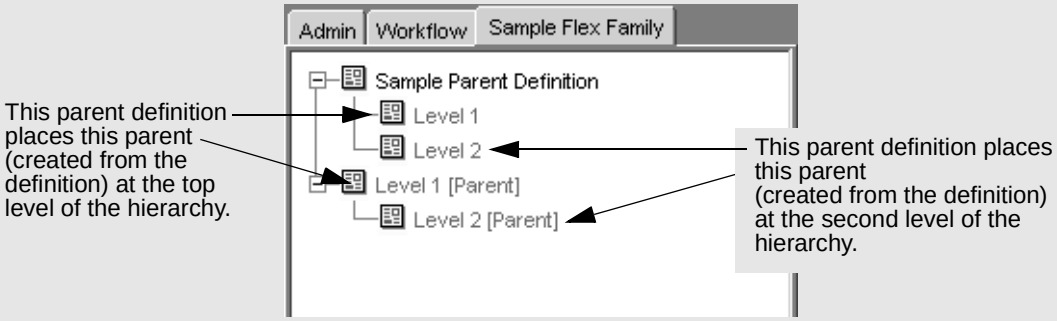
5. Go to the "Sample Flex Family" tab and expand its contents to make sure they are identical to the display below:



Note

The figure below summarizes how the objects that you created—parent definitions and parents based on the definitions—are related to each other.

Parent Definitions and Parents



6. Go to the next step.

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 Level 1 [Parent].
- The second flex definition asset will be used to place assets under Level 2 [Parent].
- 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 Sample Asset Definition**.
3. In the form that appears, fill in the fields as follows:

Field Name	Enter or Select	Comments
Name	Asset Def 1	
Sample Parent Definitions	Level 1 **Note:** Under "Single Value," click the **Required** arrow.	Choosing the Level 1 parent definition means that when you use the current flex definition to create flex assets, the assets will be placed under the parents that use the Level 1 parent definition (i.e., the assets will be placed under Level 1 [Parent]).

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 Sample Asset Definition**.
3. In the form that appears, fill in the fields as follows:

Field Name	Enter or Select	Comments
Name	Asset Def 2	
Sample Parent Definitions	Level 2 **Note:** Under "Single Value," click the **Required** arrow.	Choosing the Level 2 parent definition means that when you use the current flex definition to create flex assets, the assets will be placed under the parents that use the Level 2 parent definition (i.e., the assets will be placed under Level 2 [Parent]).

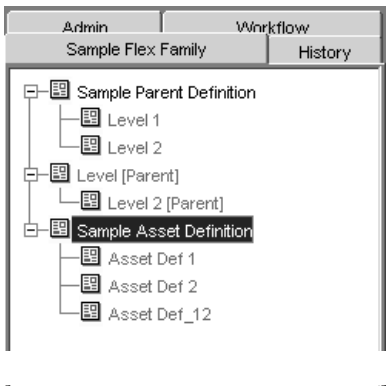
4. Click **Save**.

To create the third flex definition asset

1. From the button bar, click **New**.
2. From the list of options that appears, select **New Sample Asset Definition**.
3. In the form that appears, fill in the fields as follows:

Field Name	Enter or Select	Comments
Name	Asset Def_12	
Sample Parent Definitions	Level 1 Level 2 **Note:** Under "Single Value," click the **Required** arrow.	Choosing the Level 1 *and* Level 2 parent definitions means that when you use the current flex definition to create flex assets, the assets will be placed under the parents that use both the parent definitions (i.e., the assets will be placed under Level 1 [Parent] *and* Level 2 [Parent]).

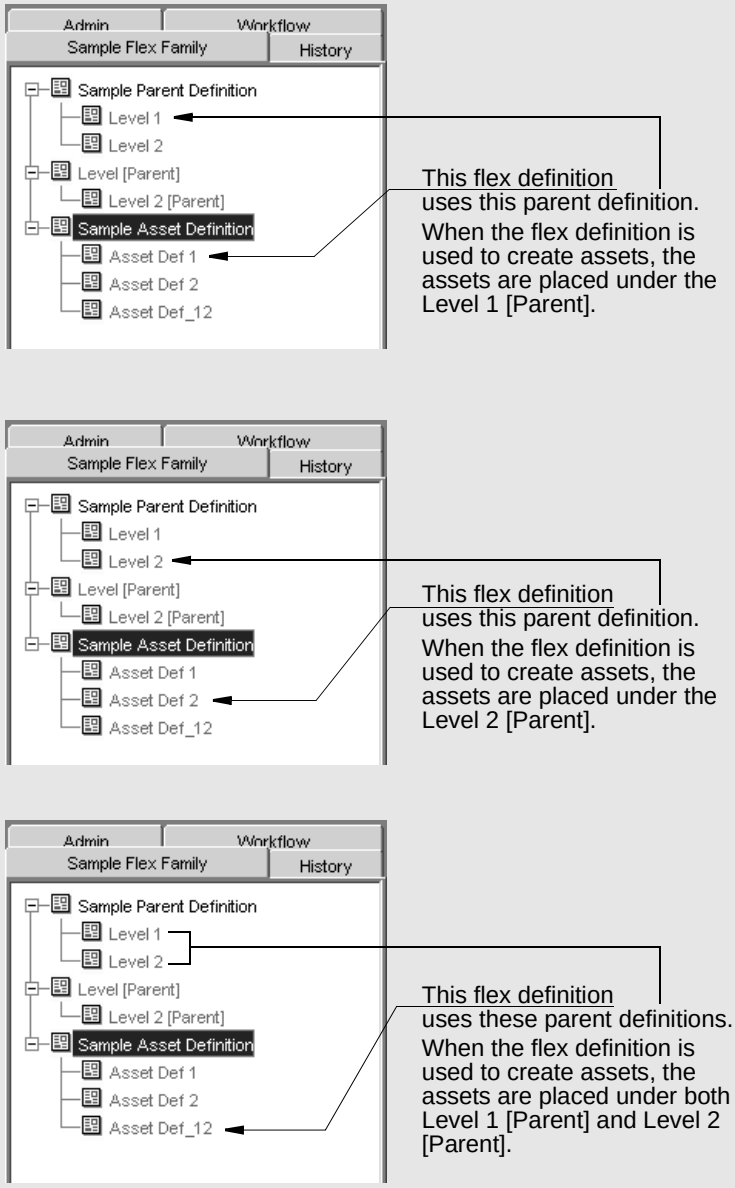
4. Click **Save**.
5. Go to the "Sample Flex Family" tab and expand its contents to make sure they are identical to the display below:



Note

The next figure summarizes how the objects that you created—flex definition assets—are related to the flex parent definitions they are based upon.

Flex Definitions and Parent Definitions



6. Go to the next step.

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 Level 1 [Parent]
- The second asset you will place under Level 2 [Parent]
- The third asset you will place under both Level 1 [Parent] *and* Level 2 [Parent].

To create the first flex asset

1. From the button bar, click **New**.
2. From the list of options that appears, select **New Sample Asset**.
3. In the form that appears, fill in the fields as follows:

Field Name	Enter or Select	Comments
Name	Asset 1	
Sample Asset Definition	AssetDef 1 **Note:** Under "Single Value," click the **Required** arrow.	Choosing AssetDef 1 means that the asset you are creating will be placed under the Level 1 [Parent].

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 Sample Asset**.
3. In the form that appears, fill in the fields as follows:

Field Name	Enter or Select	Comments
Name	Asset 2	
Sample Asset Definition	AssetDef 2 **Note:** Under "Single Value," click the **Required** arrow.	Choosing AssetDef 2 means that the asset you are creating will be placed under the Level 2 [Parent].

4. In the next form, click **Save**.

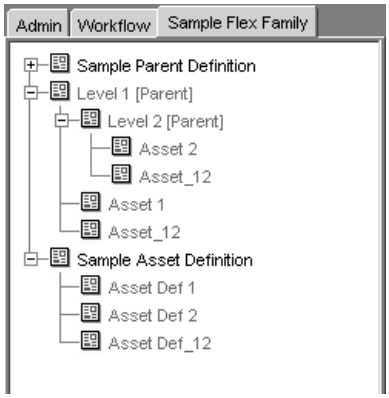
To create the third flex asset

1. From the button bar, click **New**.
2. From the list of options that appears, select **New Sample Asset**.

3. In the form that appears, fill in the fields as follows:

Field Name	Enter or Select	Comments
Name	Asset 3	
Sample Asset Definition	AssetDef 1 AssetDef 2 **Note:** Under "Single Value," click the **Required** arrow.	Choosing AssetDef 1 and AssetDef 2 means that the asset you are creating will be placed under both the Level 1 [Parent] and Level 2 [Parent].

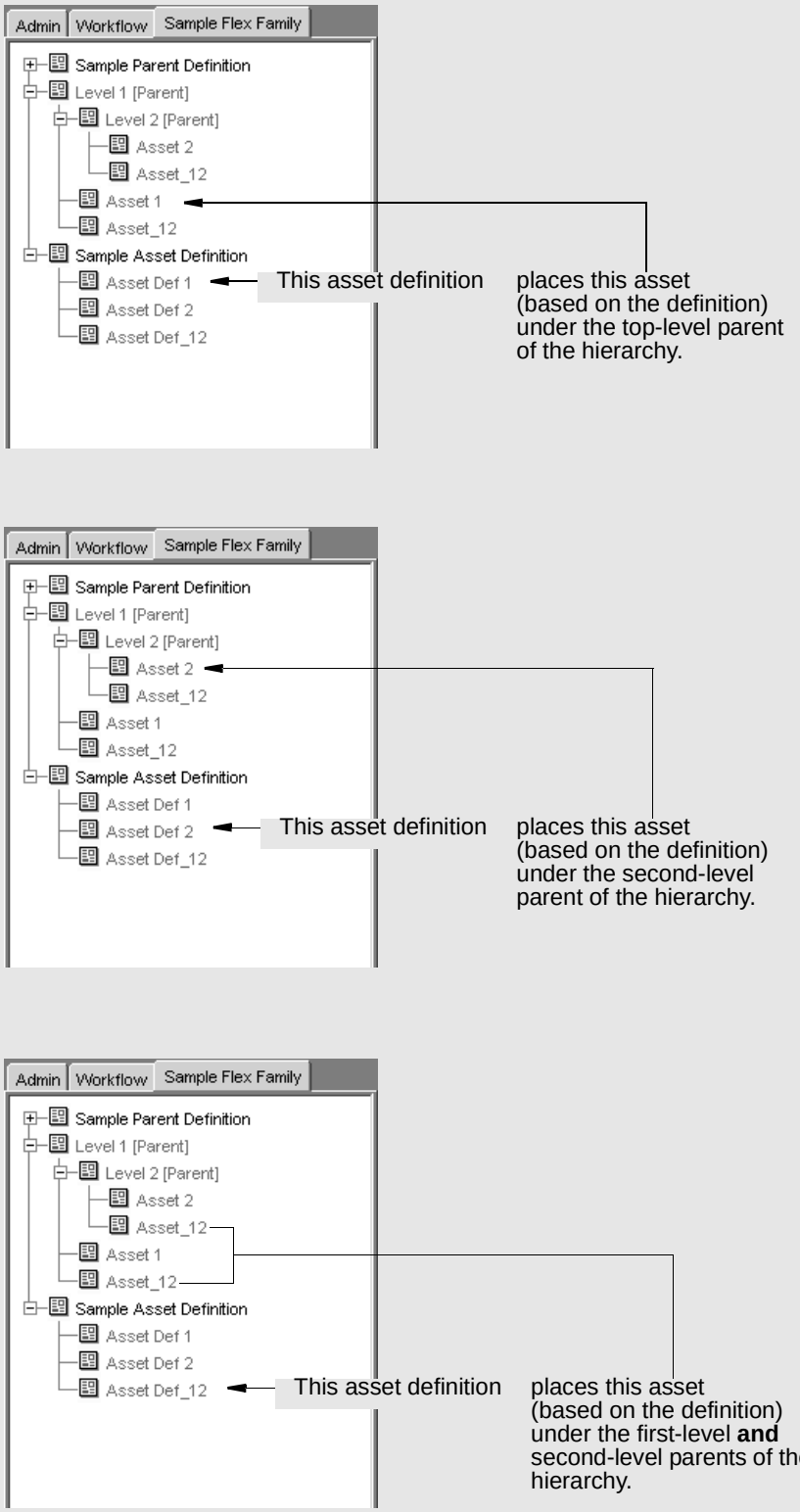
4. In the next form, click **Save**.
5. Go to the "Sample Flex Family" tab and expand its contents to make sure they are identical to the display below:



Note

The next figure summarizes how the objects that you created—assets—are related to the flex definition assets they are based upon.

Asset Definitions and Assets



Next Steps

At this point you have created a rudimentary data structure. From here, you can expand the data structure by creating multiple parents to occupy a given level, creating additional levels, 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, return to Chapter 15, “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>`) 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 713 and “Registering and Ranking Assemblers” on page 714 for more information about creating, registering, and ranking assemblers.

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.

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 on this class, see the *JavaDocs* that are located at the following URL:

   `http://e-docs.fatwire.com/CS`
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 714 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 `0` 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.

Index

A

access ID 688

accounts
  coding account management forms 580

ACL (access control list)
  and caching 113
  and live site security 576
  and site visitors 576
  identifying a visitor's ACL 578
  SiteGod 580
  SystemACL table 221

ACL tag family 576

Add Enumerated Value field 670

addData element 387

adding
  mimetypes 305
  rows to tables 240

ADF, *See* asset descriptor files

administrator rights 669

aliases 683

AMap tables 208

APIs
  Directory Services 570

AppendSelectDetails element 281

approval dependencies 508
  ASSET.LOAD 513
  ASSETSET tag family 514
  collection assets 522
  exact 510
  exists 509
  none 510
  publishing 508
  RENDER.FILTER 516
  RENDER.GETPAGEURL 515
  RENDER.LOGDEP 516

approval templates 510
  designating 471

arguments
  for Documentation Transformation filter class 207
  in XMLPost configuration files 369

article assets 190
  coding templates for 518

asset descriptor files 190
  and embedded links 291
  basic format of 274
  coding 286
  fine-tuning 298
  uploading 294

asset types
  attribute editor 188, 203
  collection 187
  core 187
  creating basic asset types 284
  creating flex asset types 316
  CSElement 187
  defined 186
  defining columns for basic asset types 276
  defining their database columns 276

deleting basic asset types 308
enabling fields for 291
naming 275
page 187
query 187
registering elements for 296
sample site 190
SiteEntry 187
template 187

ASSET.CHILDREN 521

assetid 686

AssetMaker
and embedded links 291
creating basic asset types 190, 270
in Burlington Financial 607

AssetPublication table 494

AssetPublicationTree table 194

AssetRelationTree table 191, 194, 219

assets 93
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 305
and tags 213
basic 40, 186
child 549
CS-Direct Advantage 199
database tables for basic assets 194
database tables for flex assets 208
defined 186
deleting 489
designing 185
differences between basic and flex 214
Engage 188
flex 47, 186
flex family 199
flex filter 200
getting field values 93
held 508
importing 386
indexing 212
listing children 93
loading 92
loading basic 92
retrieving field values 93
sharing 489
summary of basic and flex 214
which model to use 189

assetsets 211
and flex attribute asset types 529
and searchstates 528
attributes of type blob 534
attributes of type text 533
attributes of type url 536
creating 531
getting attribute values 532
list objects 538
overview 50, 527

AssetStubCatalog table 271

assettype column 686

AssetType table 270

Associated queries field (collection assets) 492

Association
form 191
table 191

associations 191
creating fields for 301
deleting association fields 303
flex assets 334

attribute editors 188, 203
checkbox 346
code syntax for 345
components of 342
creating 355
custom 357
editing 362
elements 353
elements for 351
eWebEditPro 347
pick asset 348
presentationobject.dtd 342
pull-down 348
radio buttons 349
textarea 350
textfield 351
XMLPost configuration file 390
XMLPost source file 390

attributes 53, 200
categories 667
creating 311
deciding how many types 311
flex filter assets 311
foreign 324
history 188, 671

inheritance of values 203
of type asset 312
visitor 188, 668

B

basic asset model
advantages 189
data structure 194
overview 190
parent-child relationships 191

basic assets
and embedded links 291

BIGINT data type 224, 277

binary visitor attributes 668, 685
and storing carts across sessions 688
collecting visitor data 683
retrieving 689

blob (binary large object) 542
BlobServer URLs 534
caching 117
code example for displaying 551
coding elements for 521
creating URLs for 89
displaying value of 535
flex attributes of type blob 534

BLOB data type 277

BlobServer 521
BLOBSERVICE tags 535
servlet 31
URL for attribute of type blob 534
URLs 543

BulkLoader
and DB2 421
command line 431
configuration file 424
configuration file sample 429
custom extraction mechanism 432
description 418
flat tables 420
IDataExtract 433
IFeedback 441
input table 421
IPopulateDataSlice 437
java interfaces 432
mapping table 423

Burlington Financial sample site
AssetMaker assets 607
bfmembers table 583
caching 610
collection assets 608
database searching 603
email to a friend page 562
home page 548
mimetypes 607
navigation elements 601
navigational bar 564
overview 44, 51, 600
plain text link 555
query assets 603, 605
section pages 553
site plan 61
today's date function 567
visitor authentication 583, 608
Wirefeed query 559

C

c variable 99

cache
and PageCriteria 120
CacheManager 120
debugging 167
the cache key 120
viewing the contents of 119

cache criteria 463, 467

cache key
overview 120

cached elements
calling 88

cacheinfo column 610

CacheManager
and mirror publishing 512
and preview 513
Satellite Server 512

caching
cache key 120
compositional dependencies 512
disable individual pages 545
flushing resultsets 265
load on database 263
overview 35
properties 266
resultsets 37, 262

calculating
approval dependencies 508

promotions 685
segments 685

calling
CSElement asset 451, 517
element 451
page name 451
SiteEntry asset 451, 517
uncached elements 88, 89

CALLSQL
example of 264

CatalogManager
servlet 31, 236
tag attributes 236

CatalogMover
command line interface 148
connecting to Content Server 142
exporting database tables 144
exporting to ZIP files 145
importing database tables 146
menu commands 143
overview 141
starting 141

catalogs
See also database tables

categories 667, 669, 675
adding 303

category
column 192, 198
field 494

Category field
collection asset 492
page asset 500
query asset 496
stylesheet asset 498

Category table 220, 270

cc.contentkey property 220

CGI
programming 101

CHAR data type 223, 277

characters, special 85

checkbox
attribute editor for flex attributes 346
input type for basic asset fields 278

child assets
basic assets 191

child nodes 565

child templates
example 555

cid variable 99

codes, status 493

coding
account creation form 579
asset descriptor files 286
attribute editor assets 345
error tracking 172, 173, 544
login forms 577
source files for XMLPost 377

coding elements
CSElement 457
error tracking 171
for basic assets 518
for blobs 521
for collections 521
for flex assets 527
for imagefile assets 521
for page assets 525
for query assets 523
for recommended assets 687
logging dependencies 513
shopping cart 688
templates 468
that collect visitor data 684
workflow elements 633

collecting visitor data 53, 683
coding pages 684
debugging 179, 689

COLLECTION argument 686

collection assets 187
and query assets 490, 495
approval dependencies 522
coding elements for 521
compositional dependencies 523
creating 491
definition 490
example code for creating links 553
example element code 549
in Burlington Financial 608
sharing 492

column constraint types 224

columns
  defining for basic asset types 276

commerce context 682
  defined 682
  tags 682

commerce ID 688

compositional dependencies 508
  and page generation 512
  ASSET.LOAD 513
  ASSETSET tag family 514
  CacheManager 512
  collection assets 523
  query assets 524
  RENDER.LOG 515
  RENDER.UNKNOWNDEPS 517

conditionals 91, 105

configuration files
  XMLPost 366
  XMLPost and flex asset types 389

connecting to Content Server 140

Constraint type field 670, 673

constraints
  adding to searchstates 540
  deleting from searchstate 540
  range 540

Content Server
  database overview 36
  session variables maintained by 154
  SOAP interface 650

Content Server context
  creating with the ftcs tag 77
  ICS object 76
  overview 76

Content Server Desktop
  overview 52

Content Server Direct
  overview 40

Content Server Direct Advantage
  asset types 199
  overview 47

Content Server Explorer 140
  creating variables in 101
  overview 140
  registering a foreign table 231
  when to use 258

Content Server Management Tools
  creating content tables 229
  creating object tables 226
  when to use 258

content tables 220
  creating 229
  managing the data in 231
  working with 236

ContentDetails element 281

ContentEditor ACL 580

ContentForm element 281

ContentServer servlet 31

cookies 157, 683
  attributes 157
  example 158
  removing 157

CookieServer servlet 31

core asset tags
  ASSET.CHILDREN 93
  ASSET.GET 93
  ASSET.LOAD 92
  ASSET.SCATTER 93

counter variables 108

createdby column 196

createddate column 196

creating 89
  assetsets 531
  association fields 301
  attribute editors 355
  basic asset types 284
  BlobServer URL embedded in HTML
  tag 542
  BlobServer URLs 90, 534, 543
  categories 303
  collection assets 491
  content tables 229
  CSElement assets 455
  entries for attribute drop-down lists 670, 673
  flex asset types 316
  flex attributes 311
  flex attributes of type asset 312
  flex definition assets 330
  flex family 316
  flex filter assets 325

flex parent assets 332
history attributes 671
history definitions 674
hyperlinks to assets in a collection 553
object tables 226
page assets 500
parent definition assets 328
product sets 337
searchstates 531
sources 304
stylesheet assets 498
template assets 465
template for element that already exists 469
tree tables 228
URLs for hyperlinks 89, 541
user accounts programmatically 580
visitor attributes 668

CS
.dtd file 84
creating the Content Server context 77
JSP actions 80
JSP declarations 81
JSP directives 81
JSP expressions 81
JSP scriptlets 81
JSP syntax 80
JSP tag libraries 82
the Content Server context 76
using directives in CS 81
XML and the FTCS tag 85
XML overview 84
XML standard beginning 84
XML version and encoding 84

CS tags
for basic and flex assets 87

cs.timeout 154

CSElement assets 187, 449
as root element for SiteEntry asset 461
calling 517
coding 457
creating 455
creating for element that already exists 459
deleting 473
editing 472
entry in ElementCatalog table 459
how to invoke one from an element 451
name to use when calling 457
previewing 473
sharing 472
where used 454

csstatus column 470

ct variable 99
used in page names 557

currentAcl session variable 154

currentUser session variable 154

custom extraction mechanism, BulkLoader 432

customizing
asset type elements 298
attribute editors 357
PostUpdate element 299
PreUpdate element 299

D

data 655
adding rows 240
coding data entry forms 240
deleting rows 243
design of 185
listing 238
querying for 238
retrieving from tables that do not hold assets 236
writing to tables that do not hold assets 236

data types 672
BIGINT 277
BLOB 277
CHAR 277
Content Server 223
CS-Direct (AssetMaker) 277
CS-Direct Advantage (flex attributes) 202
DOUBLE 277
INTEGER 277
LONGVARCHAR 277
SMALLINT 277
specifying for attribute 669
TIMESTAMP 277
valid input for 668
VARCHAR 277
web services 655, 657

database tables
AMap tables 208
AssetPublication 494
AssetRelationTree 191
bfmembers 583
content tables 220
default columns for basic asset types 195

defined 218
deleting 259
editing rows 246
exporting 144
for basic assets 190, 194
for flex assets 208
foreign tables 220
identifying the type 222
importing 146
Mimetype 305
Mungo 534
Mungo tables 208, 210
MungoBlobs 534
object tables 218
overview 36
retrieving data from non-asset tables 236
SiteCatalog 120
SitePlanTree 60
StatusCode 494
system tables 221, 232
SystemInfo 36
tree tables 219
types of 218
visitor data 683
VMVISITORALIAS 682
VMVISITORSCALARBLOB 683
VMVISITORSCALARVALUE 683
VMz 683
working with tables that do not hold assets 236

DATETIME data type 223

DB2
and BulkLoader 421

Debug Listener 174

debugging 163
promotions 689
recommendations 689
recommendations and promotions 180
session links 179, 688
visitor data collection 179, 689

DebugServer servlet 31

Default Arguments for Preview field 473

default storage directory, *see* defdir

default template 555

Default Value field 669

DefaultReader username 154

defdir 295
column 225
defined 225
specifying for new basic asset types 288

definitions, *See* flex parent definitions or flex definitions

deleteAsset element 414

deleting
assets 489
assets with XMLPost 414
association fields 303
basic asset types 308
constraint from a searchstate 540
CSElement assets 473
history attributes 671
page assets 505
rows from tables 243
SiteEntry assets 473
template assets 473

delivery system 30
user management on 570

dependencies
See also approval dependencies and compositional dependencies
approval 508
code that logs 513
compositional 508

dependency log 508

description column 195

Description field
page assets 491, 496, 498, 500
templates 466

descriptions
visitor data assets 668

development process
recommendations 678
visitor data assets 667

development system 30

DIR tag family 571

directory
entry 570
groups 570
hierarchies 570
operations 572

directory operations
  code samples 572
  error handling 574
  troubleshooting 575

Directory Services API 570

discounts 54

Document Transformation filter 327

Document Transformer Name argument 207, 327

DOUBLE data type 224, 277

drop-down fields
  examples 292
  history attributes 673

dynamic publishing
  Mirror to Server 46, 508
  overview 34

E

ED status code 493

editing
  attribute editors 362
  CSElement assets 472
  flex assets with XMLPost 412
  history attributes 671
  rows in tables 246
  SiteEntry assets 472
  template assets 472

ElementCatalog table 221

elements 521
  beginning 84
  coding 507
  customizing asset type elements 298
  editing search elements for basic asset types 306
  ending 84
  examples 548
  ExecuteQuery 492, 523, 561
  for asset types 299
  for attribute editors 351
  for flex asset types 315
  no cache criteria 450
  registering with AssetMaker 270
  required tags 84
  that work with attribute editors 342
  when use non-asset elements 455

embedded links
  and basic assets 291
  and flex assets 350
  displaying 626

enabling
  standard fields for basic asset types 291

encoding
  XML 84

encryption 577

end date 495

enddate
  column 197
  field 291

Engage
  asset types 188
  debugging 179, 180

entities 85

entry
  directory 570

enumerated lists 670, 673

errdetails variable 172

errno variable 171, 173, 544

error logging 163, 544

error tracking 171, 172, 173, 544

events
  SystemEvent table 221

eWebEditPro
  attribute editor for flex attributes 347
  creating eWebEditPro field for basic asset types 290
  input type for basic asset fields 279

exact dependency 510

ExecuteQuery element 492, 523, 561

exists dependency 509

expiration
  of cookies 158
  of sessions 154

Export to Disk publishing 46, 510
  approval templates 510

exporting

database tables 144
to ZIP files 145

externaldoctype column 198

F

field (column) types
database-specific 224

filename
column 197
field 291

Filename field
page assets 501

files
futuretense.txt 164

Filter by field 672

firewalls
and XMLPost 367

flex asset model
overview 199
parent-child relationships 200
when to use 189

flex assets
and embedded links 350
assetset examples 687
associations 334
designing 310
displaying attribute values of 530
flex asset types 48, 199
flex definitions 205
importing 386
importing with BulkLoader 420
searchstate examples 528
tag families to use 527

flex attributes 200
blob type and assetsets 534
blob type and flex filters 325
checkbox attribute editor 346
data types 202
displaying their values 530
eWebEditPro attribute editor 347
example XMLPost configuration file 393
example XMLPost source file 394
extracting values of with assetsets 532
foreign attributes 324
importing 391
inheritance of values 203
pick asset attribute editor 348
pull-down attribute editor 348
radio button attribute editors 349
text attributes and assetsets 533
text field attribute editor 351
textarea attribute editor 350
types of and assetsets 529
url type and assetsets 536

flex definitions 200
approval templates 511
as business rules 204
assigning flex filter assets to 332
create hierarchy on tree tabs 204
creating 330
deciding how many types 313
example XMLPost source file 398
importing 395

flex families 199
creating 316

flex filter assets 200, 206
assigning to flex definition assets 332
assigning to flex parent definition assets 330
creating 325
defined 207
input and output attributes 325

flex filter classes
defined 206
Document Transformation 327
registering 338

flex parent definitions 200, 203
assigning flex filter assets to 330
setting hierarchical place 314
specifying attribute inheritance 314

flex parents 200, 203
creating 332
specifying for flex asset when importing 408

FlushServer servlet
overview 39

foreign attributes 324

foreign tables 220
managing data in 232
registering 231

forms 101

full-text search 50, 212, 528

futuretense.ini file 150, 164, 170

futuretense.txt 164, 165

G

GAProductSet attribute 337

GE Lighting sample site
  content asset types 201
  overview 51
  product asset types 201

generic field (column) types 223

getting
  the children of a basic asset 521
  values from asset fields 93

groups
  directory 570

H

hash name of resultset 263

held assets 508

Hello Asset World sample site 585

HelloCS
  servlet 32

hierarchy
  directory 570

history attributes 53, 188
  creating 671

history definitions 53, 188
  creating 674
  example code 684
  retrieving data stored as 689

HTML tags
  including an XML variable in 552, 555, 566
  substituting XML variables in 104
  with BlobServer URL embedded in 542

hyperlinks 541
  creating 89

I

ICS Object
  overview 76

ics.disable cache 545

id column 195, 210

IDataExtract interface 433

identifying
  a foreign table 231
  type of database table 222
  visitors 682

IDs
  access 688
  visitor 682

IFeedback interface 441

IList object
  tags 238

ILists
  web services 655

image assets 190
  coding templates for 518

IMAGE data type 223

imagefile assets 190
  adding mimetypes for 305
  coding templates for 521
  example element code 551

implicit objects
  and CS 80

importing assets
  assets with upload fields 375
  attribute editors 390
  attributes of type URL 405
  basic assets 363
  determining how to store multiple field values 374
  flex assets 386, 420
  flex attributes 391
  flex definitions 395
  running the BulkLoader utility 431
  running XMLPost 378
  setting the asset type 369
  setting the sites 371, 372
  setting workflow 371, 372
  source file examples 377
  specifying the posting element 369

importing database tables 146

importing ZIP files 147

inheritance
  flex assets 310

iniFile session variable 155

Input Attribute Name argument 207, 328

input table
BulkLoader 421

input types
CS-Direct (basic asset types) 277

InSite Editor
and Flex Assets 628
coding for 623
INSITE.EDIT tag 626
overview 52, 624
template examples 628

Insite Editor
embedded links 626

INTEGER data type 223, 277

Inventory servlet 39, 119
introduction 119
overview 39

invoking
XMLPost 378

IPopulateDataSlice 437

J

J2EE standard and CS 29

JDBC drivers 420

JDBC-ODBC bridge 420

JSP
and variables 104
ICS Object 76
implicit objects and CS 80

JSP element
adding tag libraries 79

JSP tag libraries
changing default tag libraries 79

L

link asset
overview 187

linking sessions 682
debugging 179, 688

LIST argument 686

list file
XMLPost 379

list objects
created for assetsets 538

list variables 107
looping through 107

lists
creating 238
looping through lists 91
users in a directory 572

LISTVARNAME argument 686

loading
basic assets 92

LoadSiteTree element 282

LoadTree element 282

log files 164
futuretense.txt 164, 165

logging
dependencies 515

logging out
web site visitors 155

login forms 577

LONGVARCHAR data type 277

LOOP 107

looping through lists 107

Lower range limit field 670, 673

M

maintaining state 102

management system 30

mapping table 423

MAXCOUNT argument 686

merging property files 151

Microsoft Word
integration with 52

Mimetype field (Stylesheet) 499

MimeType table 305

mimetypes 305
in Burlington Financial 607

Mirror to Server publishing 46, 508

modifyData element 412

modular design
  examples 548

moving page assets 503

Mungo tables 208, 210, 534

MungoBlobs table 534

N

name column
  default column for all asset tables 195

Name field
  collection assets 491
  page assets 500
  query assets 496
  template assets 465

named associations 191

names
  object 92

naming
  asset types 275
  history attributes 671
  visitor attributes 668

ncode column 219

nid column 219

nodes
  child 219, 565
  IDs 565
  parent 219
  tree table 219

none dependency 510

NOT NULL constraint 224

nparentid column 219

nrank column 219

Null allowed field 669

NULL constraint 224

NUMERIC data type 224

O

object ID
  as parameter used for loading assets 92, 549

object names 92

object tables 218
  creating 226
  managing the data in 228
  working with 236

oid column 219

otype column 219

Output Attribute Name argument 207, 328

Output Document Extension argument 207, 327

ownerid column 210

P

p variable 99

page (online)
  defined 452

page assets 60, 187, 499
  coding templates for 525
  creating 500
  defined 452
  deleting 505
  moving in the site tree 503
  placing 502

page caching
  CacheManager 512
  compositional dependencies 512
  guidelines 126
  overview 35
  SystemItemCache table 221
  SystemPageCache table 221

Page Debugger 173, 179, 688
  commands 176
  continue to cursor 178
  go 178
  overview 152
  step into 177
  step out 178
  step over 177

page entries
  for SiteEntry assets 461
  for template assets 469

page name
  defined 452

PageCriteria 467
  and the Cache Key 120

pagelets
  cacheable 450
  defined 451

pagenames
  specifying in an XMLPost configuration file 369

pages
  modular design 35
  rendering 35

parametric search 50, 212

parent definitions, *see* flex parent definition assets

parents
  See also flex parent assets
  basic assets 191, 301

parser
  errors detected 85

path
  column 197
  field 291

Path field
  page assets 501

pick asset attribute editor 348

PL status code 493

Place form 502

Place Page workflow function privilege 504

placing page assets 502

posting elements
  addData 387
  debugging 383
  deleteAsset 414
  modifyData 412
  specifying in an XMLPost configuration file 369

posting, *see* importing assets

PostUpdate element 282
  customizing 299

precedence
  of variables 106

presentationobject.dtd 342

PreUpdate element 282
  customizing 299

previewing
  CacheManager 513
  CSElement and SiteEntry assets 473
  template assets 473

primary key
  content tables 220
  object tables 218

PRIMARY KEY NOT NULL constraint 224

product discounts 54

product sets 337

promotions
  calculating 685
  debugging 689
  definition 54, 188
  duration of 196
  recalculating 685

properties
  database 233
  debugging 164, 170
  in XMLPost configuration files 366
  loading property files 233
  resultsets 266
  xmldebug 158

property
  as column and field for basic asset type 270

Property Editor
  setting properties 150
  starting 150

property files
  database properties 233
  editing 150
  futuretense.ini 164, 170
  merging 151

property variables 103

proxy servers
  XMLPost 366

pubid
  defined 494
  queries 497

Publication table 62

PublicationTree table 62

publishing

approval 508
approval dependencies 508
defined 450
dynamic vs. static 34
Export to Disk 510
Mirror to Server 508
overview 32, 45

pull-down
attribute editor 348

Q

queries
code examples 246
Java examples 250, 253, 257
JSP examples 248, 252, 256
overview 262
SystemSQL table 222
tags 238
XML examples 246, 252, 254, 255

query assets 187
coding elements for 523
compositional dependencies 524
defined 492
ExecuteQuery element 492, 523
for collections 495
in Burlington Financial 603, 605
sample element code 559
sample query 493
sharing 497
templates 495

R

radio buttons
attribute editor 349
input type for basic asset fields 279

range
searchstates 540

Rank field (place pages) 503, 504

ratings
updating before calculating 685

recalculating
segments and promotions 685

recommendations
debugging 689
definition 188
flex assets 678
overview 53
process of developing 678
testing 180

referURL variable 543

registering
a foreign table 231
elements for basic asset types 296
elements with AssetMaker 270
flex filter classes 338
transformation engine 338

relationships
AssetRelationTree table 191
collections 191
parent-child (basic) 191
parent-child (flex) 200
stored as nodes 219

RemoteContentPost element 364

remotepost, *See* XMLPost

rendering
defined 450
overview 35, 45

rendermode variable 99

resargs1 101

resargs1 column 470

resdetails1 101

reserved characters 85

reserved variable names 98

Result of query field (query assets) 496

resultsets
cache flushing 265
hash names of 263
load on database 263
overview 37, 262
properties 266

revision tracking
deleting database tables 259
error numbers 173
overview 37, 46

RF status code 494

rich-text search 50, 212, 528

root element
defined 35
for addrow page 241

for page entry 469

Rootelement field (template) 469

rows
updating with CATALOGMANAGER 236

running
BulkLoader utility 431
XMLPost 378

S

Satellite Server
overview 38

Satellite servlet
caching with 115
overview 39

saving
binary data 668

scalar objects 688

scattering assets 93

scope
and variables 98
searchstates 529

search elements
editing 306

search engines
overview 47
supported 33

searches
Burlington Financial 603
database 50, 212
directory search 572
full-text 50, 212, 528
parametric 50, 212
rich-text 50, 212, 528

SearchForm element 281

searchstates 211
building 528
overview 50
range constraints 540
resultset caching 265
scope of 529
unfiltered 531
web services 655
wildcards 540

security
overview 37

segments 53
and personalization 666
calculating 685
definition 188
list of 678
overview 53
recalculating 685

seid variable 99

select
input type for basic asset fields 278

SelectSummary SQL statement 283

SelectSummarySE SQL statement 283

SELECTTO
example of 264

servlets
BlobServer 31
CatalogManager 31, 236
ContentServer 31
CookieServer 31
DebugServer 31
FlushServer (Satellite Server) 39
HelloCS 32
Inventory (Satellite Server) 39
Satellite (Satellite Server) 39
TreeManager 31

session linking 682
debugging 179, 688

session objects
list of 682

session variables 102, 154
creating 102
login and logout 155
outputting 103
setting 87, 102

sessions 154
debugging 167
example of 155
lifetime of 154
overview 37
tips 160

sharing
assets 489
collection assets 492

CSElement assets 472
query assets 497
SiteEntry assets 472
stylesheet assets 499
template assets 472

shopping carts 688
overview 51

SimpleSearch element 281

site design assets
deleting 489
sharing 489

Site Plan tab 44, 60

site tree 502

SiteCatalog table 221
page entries for templates 469

SiteEntry assets 187, 449
calling 517
creating 461
deleting 473
editing 472
how to invoke one from an element 451
previewing 473
selecting a CSElement for 461
sharing 472
where used 454

SiteGod ACL 580

SitePlanTree table 58, 62, 194
and the Site Plan tab 60
definition 219
example code that displays information from 564

sites
defined 57, 58
developing within 60
example elements 548
examples of 59
overview 44
Publication table 62
Publication Tree table 62
setting in an XMLPost configuration file 371, 372
site plan 60
SitePlanTree table 58, 62

SMALLINT data type 223, 277

SOAP
defined 650
supported version 650

SOAP tags
consuming web services 661
example 662
parameters 662

source 494
adding with asset descriptor file 289
column 277
creating for basic assets 304
field 192

source files, XMLPost
described 377
example for attribute editor 390
flex attribute 394
flex definition 398
for flex asset types 389
identifying them 379
tags 373

Source table 220

special characters 85

SQL queries
query assets 493
standard ones created for new basic asset types 299

SQL query field (query assets) 496

SSL authentication 577

start date 495

startdate
column 196
field 291

state
maintaining 102

static publishing
Export to Disk 46, 510

status 493
column 195

status codes 493

StatusCode table 220, 494

stylesheet assets 190
adding mimetypes for 305
creating 498
sharing 499

Stylesheet field 498

subtypes
  approval templates 511
  column 197

systable column 222

system design
  page caching guidelines 126

system tables 221, 232

SystemACL table 221

SystemEvent table 221

SystemInfo table 36, 221
  defdir column 225
  systable column 222

SystemItemCache table 221

SystemPageCache table 221

systems
  delivery 29
  development 29
  management 29
  testing 29

SystemSQL table 222

SystemUserAttr table 222

SystemUsers table 222

T

tables, *See* database tables

tags
  in XMLPost source files 373
  in XMLPost source files for flex assets 401

template assets 187, 449, 453, 686
  approval templates for Export to Disk 510
  coding 468
  creating 465
  default approval templates 471
  deleting 473
  displaying fields with embedded links 626
  editing 472
  entry in SiteCatalog table 469
  naming conventions 453
  previewing 473
  recommendations 687
  setting variables for 467
  sharing 472
  subtypes 511

template column 198

template field
  page asset 492, 496, 500

testing
  recommendations 180
  visitor data assets 180

testing system 30

text
  input type for basic asset fields 277
  text field attribute editor 351

text box, *See* text area

TEXT data type 223

textarea
  attribute editor for flex attributes 350
  input type for basic asset fields 278

Tile element 282

timeouts
  sessions 154, 160

TIMESTAMP data type 277

transformation engines
  defined 206
  registering 338

tree tables 219
  creating 228
  managing the data in 229
  nodes 219
  working with 237

tree tabs
  flex definitions and hierarchy 204

TreeManager
  commands 237
  servlet 31

Type field 672

types
  constraint 670, 673
  data 669, 672
  fields (columns) 223
  history type 53

U

unfiltered searchstate 531

UNIQUE NOT NULL constraint 224

UP status code 494

updatedby column 196

updateddate column 196, 494

updatetype variable 299

updating
  rows with CATALOGMANAGER 236

upload fields 278
  and XMLPost 375
  examples for basic asset types 290
  for basic asset fields 278
  stylesheet 498

uploading
  asset descriptor files 294

Upper range limit field 670, 673

URL column 225

urlexternaldoc column 198

urlexternaldocxml column 198

URLs 89
  adding variables to 100
  BlobServer 534
  Content Server log file 168
  Content Server URL 450
  for hyperlinks 541

user management
  on delivery system 570

user name 155
  default 154
  for XMLPost 370
  session variable 154

users
  account creation form 579
  USER tags 577

V

validating
  data from input forms 684

VALUE argument 686

VARCHAR data type 277

variables
  and JSP 104
  and scope 98
  appending to URLs 100
  assigning one to another 103
  best practices 106
  c 99
  cid 99
  counters 108
  creating with Content Server Explorer 101
  ct 99, 557
  displaying variable values 103
  errdetails 172
  errno 171, 173, 544
  evaluating 87, 103
  HTML forms 101
  in HTML 104
  lists 107
  outputting values 103
  p 99
  PageCriteria 467
  precedence of 106
  referURL 543
  rendermode 99
  reserved names 98
  resolving 88, 105
  retrieving variable values 103
  seid 99
  session 102
  session (see session variables)
  setting 87
  updatetype 299
  using in an HTML tag 552, 555, 566

visitor attributes 53, 188
  binary 685, 688
  creating 668
  example code 684
  retrieving 689

visitor context, defined 682

visitor data
  collecting 683
  tables 683

visitor data assets 666
  descriptions 668
  development process 667
  testing 180, 675

visitor data collection
  debugging 179, 180, 689

visitor data manager 682
  tags 682

visitor segments 53

visitors
  authentication 576, 583, 608
  collecting data for 683
  elements that collect data for 684
  identifying 682
  IDs 682

VMVISITORALIAS table 682

VMVISITORSCALARBLOB table 683

VMVISITORSCALARVALUE table 683

VMz tables 683

VO status code 494, 546

W

web services
  complex data types 657
  consuming 661
  creating custom web services 656
  defined 650
  generating client code 655
  ILists 655
  locating remote services 661
  predefined functions 654
  process flow 655
  remote procedure calls 656
  required technologies 651
  searchstates 655
  supported data types 655
  using predefined web services 654
  writing a page 657
  writing an element 658
  writing function calls 655

webservices.invoke tag 662

webservices.parameter tag 662

workflow
  and page assets 504
  assigning to imported assets 371, 372
  coding workflow elements 633
  overview 46

writing
  history definition data 683

WSDL
  defined 650
  W3C site 650

WSDL files
  Content Server 654
  creating 659
  example 659
  for remote applications 661
  generating client code 655
  location 654
  sections 659
  XML elements 661

X

XML
  and Satellite Server 116
  debugging 166, 169
  encoding 84
  entitites 85
  errors 85
  special characters 85

XML elements
  WSDL 661

XMLPost utility 300
  and flex attributes 391
  and upload fields 375
  attributes of type URL 405
  configuration file example 375
  configuration file examples 390, 397, 399, 404
  configuration files 366
  configuration properties 366
  deleting assets 414
  determining how to store multiple field values 374
  editing flex assets 412
  example source files 390, 394
  file encoding 377
  flex asset types 386
  flex definitions 395
  proxy servers 366
  running it 378
  setting the posting element 369
  setting the sites 371, 372
  setting workflow 371, 372
  source file example 377
  source file examples 398
  source file properties 371
  specifying which asset type to import 369
  tags for source files 373
  troubleshooting 383
  under program control 380
  username 370

Y

ZIP files
exporting 145
importing 147