IBM

# Customizing and Extending IBM Content Navigator

- Understand extension points and customization options
- Create an action, service, feature, and custom step processor
- Use widgets in apps, mobile development, and more

Wei-Dong Zhu
Tomas Barina
Yi Duan
Nicole Hughes
Marcel Kostal
Chad Lou
Brett Morris
Rainer Mueller-Maechler
Ron Rathgeber
Jana Saalfeld
Jian Xin Zhang
Jie Zhang


Redbooks


ibm.com/redbooks

International Technical Support Organization

# Customizing and Extending IBM Content Navigator

May 2014

SG24-8055-01

**Note:** Before using this information and the product it supports, read the information in “Notices” on page xi.

**Second Edition (May 2014)**

This edition applies to Version 2, Release 0, Modification 0 of IBM Content Navigator found in IBM FileNet Content Manager (product number 5724-R81), IBM Content Manager (product number 5724-B19), and IBM Content Manager OnDemand (product number 5724-J33).

**© Copyright International Business Machines Corporation 2012, 2014. All rights reserved.**
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

**Notices** . . . . . xi
Trademarks . . . . . xii

**Preface** . . . . . xiii
Authors . . . . . xiv
Now you can become a published author, too! . . . . . xvii
Comments welcome . . . . . xvii
Stay connected to IBM Redbooks . . . . . xviii

**Summary of changes** . . . . . xix
May 2014, Second Edition . . . . . xix

**Part 1. Introduction** . . . . . 1

**Chapter 1. Extension points and customization options** . . . . . 3
1.1 Before you begin . . . . . 4
1.1.1 IBM Content Navigator terms . . . . . 4
1.2 Development options with IBM Content Navigator . . . . . 6
1.2.1 Configuring IBM Content Navigator . . . . . 6
1.2.2 Implementing the EDS interface . . . . . 7
1.2.3 Implementing a plug-in . . . . . 9
1.2.4 Developing or integrating a custom application . . . . . 11
1.2.5 Delivering mobile access . . . . . 13
1.2.6 Common use cases and their development options summary . . . . . 14
1.3 IBM Content Navigator development architecture . . . . . 18
1.3.1 Programming interfaces . . . . . 19
1.3.2 Communication flows . . . . . 20
1.4 Developing with IBM Content Navigator APIs . . . . . 23
1.4.1 URL API . . . . . 23
1.4.2 External data services . . . . . 24
1.4.3 Plug-in API . . . . . 27
1.4.4 Content Navigator JavaScript API . . . . . 47
1.5 IBM Content Navigator samples . . . . . 55
1.5.1 Sample external data service . . . . . 55
1.5.2 Sample web pages . . . . . 55
1.5.3 Sample plug-in application . . . . . 56
1.5.4 Sample mobile application . . . . . 57
1.5.5 Sample mobile plug-in . . . . . 57
1.6 Samples we developed for this book . . . . . 58

© Copyright IBM Corp. 2012, 2014. All rights reserved. 

1.7 Conclusion . . . . . 60

**Chapter 2. Customizing desktop appearance** . . . . . 61
2.1 Customizing the desktop appearance . . . . . 62
2.2 Adding a logo and banner color . . . . . 62
2.3 Adding login notes . . . . . 65
2.4 Adding password rules . . . . . 68
2.4.1 Testing the password rules . . . . . 70
2.5 Conclusion . . . . . 72

**Chapter 3. Setting up the development environment** . . . . . 73
3.1 Prerequisites for plug-in development . . . . . 74
3.1.1 IBM Rational Application Developer . . . . . 74
3.1.2 Eclipse development environment . . . . . 75
3.1.3 WebLogic and Eclipse . . . . . 76
3.2 Setting up development environment . . . . . 76
3.2.1 Installing the Eclipse plug-in in base Eclipse environments . . . . . 77
3.2.2 Installing the Eclipse plug-in in Rational Application Developer . . . . 77
3.2.3 Troubleshooting Eclipse plug-in installation . . . . . 78
3.2.4 Verifying Eclipse plug-in installation . . . . . 78
3.3 Plug-in development . . . . . 81
3.3.1 Creating a simple plug-in project using the wizard . . . . . 81
3.3.2 Creating plug-in extension . . . . . 85
3.3.3 Packaging and building a plug-in . . . . . 90
3.3.4 Registering and testing a plug-in . . . . . 94
3.4 Creating a new empty EDS project using the wizard . . . . . 101
3.5 Getting started with the SamplePlugin . . . . . 106
3.6 Building a plug-in JAR manually . . . . . 109
3.7 Debugging plug-ins . . . . . 110
3.7.1 Remote debugging . . . . . 110
3.7.2 Local debugging . . . . . 113
3.8 Conclusion . . . . . 113

**Part 2. Customizing Content Navigator** . . . . . 115

**Chapter 4. Developing a plug-in with basic extension points** . . . . . 117
4.1 Example overview . . . . . 118
4.2 Developing the Create Dossier action . . . . . 120
4.2.1 Setting up a new plug-in project . . . . . 120
4.2.2 Packaging and deploying . . . . . 121
4.2.3 Adding the action . . . . . 122
4.2.4 Implementing the action JavaScript . . . . . 124

4.2.5 Preparing ECM system and deploying current version . . . . . 127
4.3 Developing a plug-in service to create the substructure of the dossier . . 133
4.3.1 Server type independent code: CreateSubStructureService . . . . . 136
4.3.2 IBM FileNet P8 code: CreateSubStructureServiceP8 . . . . . 138
4.3.3 IBM Content Manager code: CreateSubStructureServiceCM . . . . . 140
4.3.4 Deploying and verifying . . . . . 145
4.4 Developing the Open Dossier action . . . . . 145
4.4.1 Adding an action to the plug-in . . . . . 146
4.4.2 Extending the Action model . . . . . 147
4.4.3 Packing, deploying, and configuring . . . . . 152
4.4.4 Enhancing the Create Dossier action to also open the dossier . . . 156
4.5 Open dossier view in its own feature . . . . . 156
4.5.1 Adding the feature . . . . . 157
4.5.2 Deploying and configuring the feature . . . . . 158
4.5.3 Developing a dossier feature by using the browse feature . . . . . 159
4.6 Adding configuration . . . . . 161
4.6.1 Adding a configuration panel . . . . . 161
4.6.2 Adapting code to use configuration values . . . . . 167
4.7 Dossier management in the real world . . . . . 170
4.8 Conclusion . . . . . 170

**Chapter 5. Building a custom repository search service** . . . . . 173
5.1 Example overview . . . . . 174
5.2 Viewing results in ContentList widget . . . . . 174
5.3 Custom repository search service in sample plug-in . . . . . 180
5.3.1 Sample code of custom repository search service . . . . . 181
5.4 Query string for search against repositories . . . . . 183
5.4.1 IBM Content Manager query string . . . . . 183
5.4.2 IBM FileNet Content Manager query string . . . . . 185
5.5 Creating a new plug-in with custom repository search service . . . . . 187
5.5.1 Creating a new plug-in project . . . . . 187
5.5.2 Importing search service from sample plug-in . . . . . 188
5.5.3 Importing feature pane from sample plug-in . . . . . 194
5.5.4 Building and deploying the new plug-in . . . . . 197
5.6 Adding a new function to the existing search service . . . . . 199
5.6.1 Adding a paging function . . . . . 199
5.6.2 Getting the result properties setting from admin configuration . . . . 207
5.6.3 Main files of custom search plug-in . . . . . 210
5.7 Conclusion . . . . . 210

**Chapter 6. Creating a feature with search services and widgets** . . . . . 211
6.1 Example overview . . . . . 212
6.2 Adjusting the layout of the feature . . . . . 213

6.3 Creating a tree widget to show a custom tree . . . . . 218
6.4 Adding function on the class node to search and show results. . . . . 223
6.5 Configuring more modules to the ContentList widget . . . . . 225
6.6 Conclusion. . . . . 231

**Chapter 7. Implementing request and response filters and external data services** . . . . . 233
7.1 Request and response filter overview . . . . . 234
7.1.1 When to implement request and response filters . . . . . 234
7.2 External data service (EDS) overview. . . . . 235
7.2.1 When to use EDS . . . . . 235
7.2.2 Sample EDS service provided with IBM Content Navigator . . . . . 239
7.3 Example overview . . . . . 240
7.4 Setting up the sample EDS project . . . . . 242
7.5 Choice lists . . . . . 246
7.5.1 Simple choice lists. . . . . 246
7.5.2 Dependant choice lists . . . . . 255
7.6 Property field validation . . . . . 256
7.7 Setting field properties. . . . . 257
7.7.1 Setting field status. . . . . 257
7.7.2 Setting field minimum and maximum values. . . . . 259
7.8 Ordering properties on the add dialog. . . . . 260
7.9 Filter for large choice lists . . . . . 266
7.10 Deployment and configuration. . . . . 274
7.11 Conclusion. . . . . 276

**Chapter 8. Creating a custom step processor** . . . . . 277
8.1 Workflow step processors overview . . . . . 278
8.2 IBM Content Navigator step processors . . . . . 278
8.3 Custom step processors . . . . . 281
8.4 Example overview . . . . . 281
8.5 Creating a custom step processor. . . . . 282
8.5.1 StepProcessorRedbkLayout.html . . . . . 282
8.5.2 StepProcessorRedbkLayout.js . . . . . 283
8.5.3 StepProcessorRedbkLayout.jsp . . . . . 285
8.6 Deploying the custom step processor . . . . . 287
8.7 Registering the custom step processor. . . . . 287
8.8 Configuring Application Space and in-basket . . . . . 288
8.9 Adding an action to use an embedded viewer . . . . . 289
8.9.1 Updating StepProcessorAction.java . . . . . 290
8.9.2 Updating StepProcessorActionPlugin.js . . . . . 290
8.9.3 Building and deploying the action plug-in . . . . . 291
8.10 Using and testing the custom step processor . . . . . 292

8.10.1 Creating the workflow with custom step processor . . . . . 292
8.10.2 Adding subscription to initiate workflow . . . . . 293
8.11 Adding external data services . . . . . 293
8.11.1 Testing the workflow and custom step processor . . . . . 294
8.12 Conclusion. . . . . 296

**Chapter 9. Using Content Navigator widgets in other applications** . . . . 297
9.1 Integration into other applications overview . . . . . 298
9.2 Example overview . . . . . 298
9.3 Externalizing IBM Content Navigator widgets . . . . . 299
9.3.1 Approaches of IBM Content Navigator externalization . . . . . 299
9.3.2 Simulation of IBM Content Navigator integration part 1 . . . . . 303
9.4 Integrating Content Navigator with URL API . . . . . 308
9.5 Integrating Content Navigator with a specific feature . . . . . 310
9.6 Integrating Content Navigator with a specific layout . . . . . 313
9.6.1 Setting up a new plug-in project . . . . . 313
9.6.2 Adding the layout . . . . . 314
9.6.3 Adding the ContentList widget to the layout . . . . . 320
9.7 Integrating specific widgets of Content Navigator . . . . . 324
9.7.1 Initialization phase of Content Navigator. . . . . 325
9.7.2 Step 1: Initialize Dojo and Content Navigator libraries . . . . . 328
9.7.3 Step 2: Select the appropriate visual widget . . . . . 331
9.7.4 Step 3: Select and initialize the necessary model classes . . . . . 337
9.7.5 Step 4: Wire the widgets together through event registration . . . . . 342
9.8 Integrating the externalized widgets (Step 5) . . . . . 344
9.8.1 Deploying the externalized widget (unbound integration) . . . . . 346
9.8.2 Setting up external widgets in the container (bound integration) . . 347
9.8.3 Defining a reverse proxy in an HTTP Server . . . . . 347
9.8.4 Outline of a bound integration . . . . . 349
9.9 Integrating stand-alone widgets in a Microsoft SharePoint page . . . . . 356
9.9.1 Example overview . . . . . 356
9.9.2 Implementing the Microsoft SharePoint integration . . . . . 356
9.10 Integrating stand-alone widgets as a portlet in WebSphere Portal . . . . 359
9.11 Conclusion. . . . . 364

**Chapter 10. Customizing built-in viewers and integrating third-party viewers** . . . . . 365
10.1 Built-in viewers overview . . . . . 366
10.2 Example overview . . . . . 370
10.3 Customizing the FileNet Viewer . . . . . 370
10.3.1 Adding headers and footers to documents for branding and confidentiality requirements . . . . . 371
10.3.2 Making annotations anonymous . . . . . 373

10.3.3 Disabling document streaming . . . . . 375
10.4 Exploring viewer extensibility . . . . . 376
10.4.1 Viewer architecture . . . . . 376
10.4.2 Extension points . . . . . 377
10.5 Integrating third-party viewers into IBM Content Navigator . . . . . 387
10.5.1 Snowbound VirtualViewer . . . . . 387
10.5.2 Informative Graphics Brava! Enterprise Viewer . . . . . 407
10.6 Conclusion . . . . . 412

**Chapter 11. Extending solutions to mobile platform** . . . . . 413
11.1 IBM Content Navigator mobile solutions . . . . . 414
11.1.1 Mobile development options . . . . . 414
11.2 Example overview . . . . . 417
11.3 Customizing IBM Worklight sample . . . . . 419
11.3.1 Example overview . . . . . 420
11.3.2 Environment preparation . . . . . 422
11.3.3 Updating model layer . . . . . 425
11.3.4 Developing views . . . . . 426
11.3.5 Adding controller for WorkView . . . . . 427
11.3.6 Integrating to sample . . . . . 430
11.3.7 Packaging and deployment . . . . . 432
11.4 Conclusion . . . . . 435

**Chapter 12. Extending Profile Plug-in for Microsoft Lync Server** . . . . . 437
12.1 What is available in IBM Content Navigator 2.0.2 . . . . . 438
12.2 Microsoft Lync Server and UCWA . . . . . 440
12.2.1 UCMA . . . . . 440
12.2.2 UCWA . . . . . 440
12.3 Example overview . . . . . 441
12.4 High-level design . . . . . 442
12.4.1 Goals of the new plug-in . . . . . 442
12.4.2 Contact card or business card function . . . . . 443
12.4.3 Microsoft Lync Contact Card and Status Service . . . . . 443
12.5 Implementing the Lync Plug-in . . . . . 444
12.5.1 Configuring the Lync Plug-in . . . . . 444
12.5.2 Microsoft Lync Service user interface . . . . . 445
12.5.3 Error handling . . . . . 446
12.5.4 Log in to Microsoft Lync Service . . . . . 447
12.5.5 Getting contact information from Microsoft Lync Server . . . . . 448
12.5.6 Adding a response filter to user name fields . . . . . 450
12.6 Object-oriented design for Java and JavaScript . . . . . 452
12.6.1 Java objects of the Lync Plug-in . . . . . 452
12.6.2 Extending the existing Java classes . . . . . 452

12.6.3 Extending JavaScript classes . . . . . 455
12.6.4 Extending the configuration pane . . . . . 456
12.6.5 Notes for programming with Microsoft Lync UCWA Service . . . . . 456
12.6.6 Displaying business card . . . . . 457
12.7 Setup, installation, and enhancements . . . . . 459
12.7.1 Considerations for development environment . . . . . 459
12.7.2 Performance . . . . . 459
12.7.3 Debugging the plug-in application . . . . . 459
12.7.4 Installing and setting up the Lync Plug-in . . . . . 460
12.7.5 Future enhancements . . . . . 461
12.8 Conclusion . . . . . 462

**Part 3. Deployment, debugging, and troubleshooting** . . . . . 463

**Chapter 13. Component deployment** . . . . . 465
13.1 Strategies for managing a production environment . . . . . 466
13.2 Identifying the components of your application . . . . . 467
13.2.1 Component hierarchy . . . . . 467
13.2.2 Persistence characteristics . . . . . 469
13.2.3 Database layer . . . . . 469
13.3 Manual deployment of an IBM Content Navigator solution . . . . . 470
13.3.1 Repository dependencies . . . . . 470
13.3.2 IBM Content Navigator desktop . . . . . 471
13.4 Injecting a level of predictability in your environment . . . . . 478
13.5 Using export and import to deploy a Content Navigator solution . . . . . 481
13.5.1 Using the Export function within the Administration Desktop . . . . 482
13.5.2 Using the Import function within the Administration Desktop . . . . 488
13.5.3 Import summary and reports . . . . . 492
13.6 Conclusion . . . . . 493

**Chapter 14. Debugging and troubleshooting** . . . . . 495
14.1 Client debugging . . . . . 496
14.1.1 Firefox . . . . . 496
14.1.2 Chrome . . . . . 497
14.1.3 Fiddler . . . . . 497
14.2 Client logging . . . . . 498
14.2.1 Logging levels and browser types . . . . . 498
14.2.2 Enabling logging in JavaScript files . . . . . 499
14.3 Server-side logging . . . . . 500
14.4 Log and trace files . . . . . 504
14.4.1 IBM Content Navigator log files . . . . . 504
14.4.2 Application server log files . . . . . 505
14.5 Troubleshooting . . . . . 505
14.5.1 IBM Content Navigator troubleshooting tools . . . . . 506

14.6 Conclusion . . . . . 510

**Part 4. Appendixes** . . . . . 511

**Appendix A. Action privileges** . . . . . 513

**Appendix B. Document class definition** . . . . . 515
Adding example class to IBM FileNet Content Manager . . . . . 516
Adding example class to IBM Content Manager . . . . . 517

**Appendix C. Core code for the custom search plug-in project** . . . . . 519
VirtualFolderBrowsePane.js . . . . . 520
Enhanced SamplePluginSearchServiceP8.java code . . . . . 528

**Appendix D. Additional material** . . . . . 535
Locating the web material . . . . . 535
Using the web material . . . . . 536
System requirements for downloading the web material . . . . . 536
Downloading and extracting the web material . . . . . 536

**Related publications** . . . . . 537
IBM Redbooks . . . . . 537
Online resources . . . . . 537
Help from IBM . . . . . 538

# Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:
*IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.*

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

| | | |
|---|---|---|
| AIX® | ImagePlus® | Sametime® |
| DB2® | Lotus® | WebSphere® |
| Domino® | Rational® | Worklight® |
| FileNet® | Redbooks® | |
| IBM® | Redbooks (logo) ® | |

The following terms are trademarks of other companies:

Adobe, the Adobe logo, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Worklight is trademark or registered trademark of Worklight, an IBM Company.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

# Preface

IBM® Content Navigator provides a unified user interface for your Enterprise Content Management (ECM) solutions. It also provides a robust development platform so you can build a customized user interface and applications to deliver value and an intelligent, business-centric experience.

This IBM Redbooks® publication guides you through the Content Navigator platform, its architecture, and the available programming interfaces. It describes how you can configure and customize the user interface with the administration tools provided, and how you can customize and extend Content Navigator using available development options with sample code. Specifically, the book shows you how to set up a development environment, and develop plug-ins that add an action, service, and feature to the user interface. Customization topics also include implementing request and response filters, external data services (EDS), creating custom step processors, and using Content Navigator widgets in other applications. In addition, this book covers mobile development, viewer customization, component deployment, and debugging and troubleshooting.

This book is intended for IT architects, application designers, and developers who work with IBM Content Navigator and IBM ECM products. It offers both a high-level description of how to extend and customize IBM Content Navigator and also more technical details of how to do implementation with sample code.

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# Authors

This book was produced by a team of specialists from around the world working in Seattle and at the IBM Software Development lab in Kirkland, Washington.

**Wei-Dong Zhu** (Jackie) is an Enterprise Content Management (ECM) Project Leader with the International Technical Support Organization (ITSO). Jackie joined IBM in 1996 and has more than 10 years of software development experience in accounting, image workflow processing, and digital media distribution. She is a Certified Solution Designer for IBM Content Manager, and has managed many Enterprise Content Management Redbooks publications. Jackie holds a Master of Science degree in Computer Science from the University of Southern California.

**Tomas Barina** is an ECM Consultant with the IBM Software Group in Czech Republic. He has more than 10 years of experience in the content management field. For the last eight years, his focus has been primarily on design and delivery of IBM FileNet® solutions. His areas of expertise include solution design, ECM, and mobile development. Tomas holds master's degree in Computer Science and Artificial Intelligence from Czech Technical University.

**Yi Duan** is an ECM Advisory Software Engineer with the IBM Software Group in China. He has over 11 years of experience in the software engineering field. Yi joined the IBM Content Navigator Quality Assurance team with the first release in 2011. He has extensive experience with IBM Content Navigator, especially in EDS. Before joining the team, Yi worked with the IBM Document Manager team for six years. Yi holds a master's degree in Computer Science from Beijing University of Posts and Telecommunications.

**Nicole Hughes** is a Certified IT Specialist with the North America ECM Channel Sales team. Nicole specializes in IBM Content Analytics with Enterprise Search and IBM Content Classification and is also skilled in IBM Content Manager, IBM FileNet, and IBM Lotus® Domino® solutions. Nicole joined IBM in 2003 through the Green Pasture Software acquisition and has 17 years of experience in the ECM industry. Her expertise in system design has led to successful implementations of ECM solutions, such as facilities management, engineering drawings management, and procedures/policies management in various industries including manufacturing and utilities. Nicole holds a B.A. degree from Washington State University.

**Marcel Kostal** is an ECM Solution Consultant with IBM Software Group in Slovakia. He has more than 12 years of experience in designing, developing, and delivering Java Platform, Enterprise Edition solutions. For the past five years, Marcel has focused on ECM solutions primarily in the banking sector. His areas

of expertise include solution architecture, application design, implementation, integration, and technical enablement with the IBM FileNet P8 product suite.

**Chad Lou** is a Senior Software Engineer with the IBM Software Group. He works extensively in designing and implementing solutions in the ECM area. Chad has worked on various platforms, with Web 2.0 technology at the forefront, Java in the middle-tier, and database in the back end. Before joining IBM, Chad worked at Twentieth Century Fox as a Technical Lead.

**Brett Morris** is the Lead Architect for Enterprise Content Management Client Development and one of the primary people responsible for the architecture and design of IBM Content Navigator. He has been with IBM for nearly fifteen years and has over a decade of experience in Enterprise Content Management. Brett has written several technical articles and conducts sessions at many technical conferences.

**Rainer Mueller-Maechler** is a Senior IT Specialist with the IBM Software Group in Zurich, Switzerland. He has ten years of software engineering experience in the ECM field including consulting, design, development, installation, and integration. He has been with IBM for three years; previously, he worked for several IBM FileNet Business Partners for seven years. Rainer has strong background in client development for ECM systems and for five years has contributed to one of the leading ECM client products. Rainer holds a master's degree in Computer Science from the University of Ulm.

**Ron Rathgeber** is an ECM Software Architect working on the IBM Enterprise Records product with the IBM Software Group in United States. Previously, he worked on the IBM FileNet Capture and FileNet WorkForce Desktop products. Ron has over 20 years of experience in the imaging and content management fields, working in the software development group. Ron holds a Bachelor of Science degree in Information and Computer Science from the University of California, Irvine.

**Jana Saalfeld** is a Certified IT Specialist with the IBM Software Group Services in Germany. Jana has eight years of experience in the ECM field and has been with IBM for nine years. She holds a bachelor's degree in Applied Computer Science at the Cooperative State University of Stuttgart. Her areas of expertise include solution and application design, implementation, integration, and technical enablement within the ECM product suite.

**Jian Xin Zhang** is a Software Developer for IBM Content Navigator with the IBM Software Group in China. Jian joined IBM in 2004 and he has more than 10 years of software development experience. He has over nine years of development experience on IBM Content Manager and IBM FileNet products. Jian holds a master's degree in Computer Science from Beijing Institute of Technology.

**Jie Zhang** is a Software Developer for IBM Content Navigator in the China Development Lab. Jie joined IBM in 2005 and he has more than 10 years of software development experience. He has over six years experience on IBM Content Manager and IBM FileNet products including quality assurance and development. Jie holds a master's degree in Computer Science from Tsinghua University.

Special thanks to the IBM Content Navigator management team and lead architects. Without their support and leadership, we could not have completed the book:

IBM Content Navigator product management team:
Ian Story
Scott Mills

IBM Content Navigator development and testing management team:
Dana Morris
Mary Booher
Song Guan
Hansen Lee
Kristin Wallio

IBM Content Navigator lead architect:
Brian Owings

Thanks to the following people for their contributions to this project:
Julia Bamford
Yoav Ben-Yair
Matt Bogusz
Michael Davidovich
Jon Elslip
James Iuliano
Bob LaTurner
Oren Paikowsky
Robert Rusch
Eitan Schreiber
IBM Software Group, US, China, and Israel

Alen Dranikov
George Farnham
Informative Graphics and Snowbound Software

Thanks to the authors of the first edition of this book, which was published October 2012:
Wei-Dong Zhu (Jackie), Ya Zhou Luo, Johnny Meissler, Ron Rathgeber, Harry Richards, Jana Saalfeld, Sebastian Schmid, and Kosta Tachtevrenidis.

## Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

**ibm.com**/redbooks/residencies.html

## Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  **ibm.com**/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

## Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

# Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition might also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-8055-01
for Customizing and Extending IBM Content Navigator
as created or updated on April 13, 2015.

## May 2014, Second Edition

This second edition has extensive revisions because of changes in IBM Content Navigator software and the feedback we received from the previous edition of the book. When possible, more explanations are provided regarding why, where, and how to do customization and extensions for Content Navigator. We created and documented many new samples based on common requests.

Specifically, this edition reflects the addition, modification, or deletion of information as described here. The previous edition is downloadable with this second edition book.

### New information

The following chapters are added to the book or have gone through major changes for this second edition:

- Chapter 1, “Extension points and customization options” on page 3 (major changes)

  Major revisions include better explanations of Content Navigator architecture, extension points, and customization options. To help you to better prepare for doing customizations, we added two new sections:

  - 1.1, “Before you begin” on page 4
  - 1.6, “Samples we developed for this book” on page 58

- Chapter 3, “Setting up the development environment” on page 73 (major changes)

  Major revisions include adding a new procedure for setting up the development environment. The chapter also shows how to create a new

© Copyright IBM Corp. 2012, 2014. All rights reserved.

plug-in project by using an Eclipse wizard and includes references for testing and debugging your application.

- Chapter 4, “Developing a plug-in with basic extension points” on page 117 (new chapter)
- Chapter 5, “Building a custom repository search service” on page 173 (new chapter)
- Chapter 6, “Creating a feature with search services and widgets” on page 211 (new chapter)
- Chapter 8, “Creating a custom step processor” on page 277 (new chapter)
- Chapter 9, “Using Content Navigator widgets in other applications” on page 297 (major changes)

  Major reworking of the chapter content includes better explanations, approaches, logic, and flow, and includes sample code.
- Chapter 11, “Extending solutions to mobile platform” on page 413 (major changes)

  Major reworking was done to the chapter based on the new mobile application sample.
- Chapter 12, “Extending Profile Plug-in for Microsoft Lync Server” on page 437 (new chapter)

## Changed information

The following chapters have been updated with latest features and functions of Content Navigator and new sections added to cover the additional information:

- Chapter 2, “Customizing desktop appearance” on page 61

  This is a recap of section 3.6 from the previous edition of the book.
- Chapter 7, “Implementing request and response filters and external data services” on page 233

  Improved explanation of when to use external data services (EDS) and when to implement request and response filters.
- Chapter 10, “Customizing built-in viewers and integrating third-party viewers” on page 365

  Added a section about integrating Informative Graphics Brava! Enterprise Viewer in addition to the original Snowbound VirtualViewer section.

- Chapter 13, “Component deployment” on page 465

  Added a section about how to use the new export and import tools for solution deployment.

- Chapter 14, “Debugging and troubleshooting” on page 495

  Added a section about tools available for debugging and troubleshooting.

## Deleted information

Since the previous edition of the book, the following chapters were removed. The reason is because we want to focus on customizing and extending Content Navigator, and not cover the general introduction and concepts that are in the IBM Knowledge Center.

- Chapter 1, “Introducing IBM Content Navigator”
- Chapter 2, “Functions, features, and concepts”
- Chapter 3, “Configuration and customization”

  Section 3.6 from this chapter has been retained because of popular demand; but, it is renumbered as Chapter 2 in this second edition of the book.

- Chapter 6, “Developing a plug-in to add an action”

  This chapter is replaced with the new chapter, Chapter 4. “Developing a plug-in with basic extension points of IBM Content Navigator” which covers developing a plug-in that adds multiple actions, services, and features.

## Previous edition is downloadable

The previous edition of the book is still downloadable as additional material with this second edition of the book. See Appendix D, “Additional material” on page 535 for information.

# Part 1

# Introduction

In this part, we introduce the extension points and customization options for working with IBM Content Navigator, and how to set up a development environment for the customization. In addition, we show how to customize desktop appearance without any coding. This part contains the following chapters:


- Chapter 1, “Extension points and customization options” on page 3
- Chapter 2, “Customizing desktop appearance” on page 61
- Chapter 3, “Setting up the development environment” on page 73


© Copyright IBM Corp. 2012, 2014. All rights reserved.

1

# Extension points and customization options

IBM Content Navigator provides a unified user interface for your Enterprise Content Management solutions. It also provides a robust development platform that you can use to build a customized user interface and applications to deliver value and an intelligent, business-centric experience. This chapter guides you through this platform, the architecture of the application, and available programming interfaces to help you understand the available development options.

This chapter covers the following topics:

- Before you begin
- Development options with IBM Content Navigator
- IBM Content Navigator development architecture
- Developing with IBM Content Navigator APIs
- IBM Content Navigator samples
- Samples we developed for this book

© Copyright IBM Corp. 2012, 2014. All rights reserved.

## 1.1 Before you begin

To understand this book, you must first understand the terminology. To understand how to develop your user interface of your solution, you must know what are the available options. This section defines several important terms that are used in this book. 1.2, “Development options with IBM Content Navigator” on page 6 describes the options you can use to fulfill business requirements.

### 1.1.1 IBM Content Navigator terms

Several terms used in this book are unique to IBM Content Navigator or have special meaning within the context of IBM Content Navigator administration and development.

- Repository: A *repository* in this book represents a document management system to which IBM Content Navigator can connect. It can include one or more of these systems: IBM FileNet Content Manager, IBM Content Manager, IBM Content Manager OnDemand, or any third-party document management system that implements Content Management Interoperability Standard (CMIS).
- Content item: The term *content item* is used to address documents, folders, custom objects and individual items stored in various repositories to which IBM Content Navigator can connect.
- Desktop: An IBM Content Navigator *desktop* represents a tailored view of the available repositories and the functionality that IBM Content Navigator can provide. The desktop is one instance of the IBM Content Navigator application within the Content Navigator deployment, for which you can configure a different appearance, different available functionality, and different repositories that will be available and visible within the user experience. A common real-world scenario is that one desktop is dedicated to one business role or department.
- Plug-in: IBM Content Navigator *plug-in* is a powerful mechanism that allows you to extend and change the capabilities of IBM Content Navigator to meet the specific business requirements of your application without modifying the code of the software itself. IBM Content Navigator plug-in is a Java class that a developer must extend to implement a custom plug-in; in most cases it refers to a Java archive (JAR) file, containing Java classes that implement defined middle tier (midtier) extension points and also web resources for extending a browser UI.

- Extension point: The term *extension point* in this book is used to address one of the available options you can implement in your plug-in, to add new or change existing functionality. It is primarily represented as a Java class that is referenced by your Plugin class.
- Feature: An IBM Content Navigator *feature* refers to top-level menu item that provides basic functionality in IBM Content Navigator such as browse, search, or favorites. It also refers to one of available *extension points* in plug-in development.
- Layout: The term *layout* is used in two ways in this book. Either it refers to an *extension point*, that allows you modify the overall look and feel of IBM Content Navigator or it refers to a *dijit layout*. (See the term *dijit* below)
- Content Navigator API: A *Content Navigator API* is a set of Java and JavaScript classes you can use to develop a plug-in or to implement your own application leveraging IBM Content Navigator functionality.
- External data services: The *external data services* (EDS) is an IBM Content Navigator plug-in that allows you to modify property behavior by introducing new REST interface. EDS is designed as a quick and easy starting point for your customization, without the need to implement a plug-in.
- Widget: A *widget* is a user interface component to provide a specific function that can be used in constructing a user interface. Widgets are typically packaged in a library of related components. A designer can construct a web page from multiple widgets that work together to provide a unified visual experience to users. A widget can be a visual widget, which is responsible for a portion of the interface that users see such as a tree view, or a non-visual widget, which provides supporting functionality to the user interface, such as form validation.
- Dojo: A *dojo* is a JavaScript library designed to increase speed and ease of development of client-side code for a web application. Dojo provides a library of widgets that can be used to easily construct a client-based user interface for a web application. Dojo widgets are designed to be both configurable and extensible.
- Dijit: The word *dijit* is an abbreviation for a Dojo widget.
- Dijit layout: A *dijit layout* is a dijit that contains child widgets. It is a JavaScript class that extends dijit/_Container class or its subclasses.
- Step processor: A *step processor* in IBM Content Navigator is a widget that displays a human task in a workflow. It extends the ecm/widget/process/ProcessorLayout dijit.

## 1.2 Development options with IBM Content Navigator

IBM Content Navigator provides several options to customize or develop the user experience for your application. Each option requires a different development skill set, amount of effort, and time to achieve the goal. Therefore, it is important to understand what each option was designed for and how each option might best fit within your project and application requirements.

The options are as follows:

- Configuring IBM Content Navigator
- Implementing the EDS interface
- Implementing a plug-in
- Developing or integrating a custom application
- Delivering mobile access

The following sections describe each option, the skills required to develop or implement these options, and the links that can help you learn more about these topics. In 1.2.6, “Common use cases and their development options summary” on page 14, we provide a list of common use cases and summarize where you want to look for customization and extension.

### 1.2.1 Configuring IBM Content Navigator

The easiest way to change the appearance of the IBM Content Navigator desktop is through configuration. In most cases, this is often the most common option used to comply with corporate naming and visual standards of an organization. IBM Content Navigator contains an Administration Desktop that you can use to configure several visual aspects of the desktop:

- Specify the name of the desktop.
- Add a company logo to both the login page and the banner of the desktop.
- Alter the colors used in the desktop banner.

In addition, on the global level (for all desktops) you can configure the following visual aspects:

- Change icons for specific MIME types.
- Alter the labels used in the desktop.

Configuration options in IBM Content Navigator can also cover some security and business requirements of the overall solution. Desktop configuration can cover the following requirements:

- Add useful notes to the login page such as “forgot password” information.
- Add or remove features from the desktop.
- Alter the menu items on both toolbar and context menus.
- Specify the displayed properties for which you can configure:
  - System properties that will be displayed to the user.
  - Properties that will be displayed for documents and folders in the content list when users are browsing.
  - Properties available in search operations, together with available operators for specific data type.

### Required skills

The configuration tasks within IBM Content Navigator can all be performed as administrative tasks through the Administration Desktop and do not require development or coding to complete. For more information about configuring IBM Content Navigator desktop, see Chapter 2, “Customizing desktop appearance” on page 61.

### Helpful links

See information about defining desktops in the IBM Knowledge Center to learn more about configuring Content Navigator:

http://pic.dhe.ibm.com/infocenter/cmgmt/v8r4m0/index.jsp?topic=%2Fcom.ibm.installingeuc.doc%2Feucco023.htm

## 1.2.2 Implementing the EDS interface

The underlying repository of your solution has specific options for how properties are defined in your document model and system taxonomy. At a minimum, it allows you to specify the data type. The system may also allow you to specify data validations such as required status or read-only status. However, you cannot change them during run time, which is a common requirement in complex solutions including workflows and various document and folder states. Another common requirement is the ability to predefine values that a property can hold. As is, you may only be able to define a static choice list within the repository without any relation or connection to an external system. EDS interface was introduced to address these issues in a light-weight manner, without the need for you to understand Java or implement an IBM Content Navigator plug-in.

An external data service is invoked for actions such as adding and editing content items, creating and using searches, and editing step processor properties in IBM Content Navigator or IBM Content Navigator for Microsoft Office. The following use cases can be covered by implementing an EDS REST interface:

- Prefill properties with values.
- Look up the choice list values for a property or dependent property.
- Set minimum and maximum property values.
- Set property status or controls, such as read-only, required or hidden.
- Implement property validation and error checking.

Your implementation of EDS REST interface is invoked by IBM Content Navigator (EDS plug-in) without propagation of a user's session and credentials. EDS plug-in sends only the user ID in the ClientContext node of the JavaScript Object Notation (JSON) request. EDS is therefore not suitable to implement, if you need this credential to connect to your external system, for example to filter the results (choice list values) by logged in user.

**Tip:** If the use cases or the REST interface does not cover your needs, you can implement your own plug-in for request and response filters. The external data services are implemented as one case for the request and response filters in IBM Content Navigator plug-in.

## Required skills

You need these skills to implement the external data service REST interface:

- Web application development (GET and POST request)
- Read and write JSON

See 1.4.2, "External data services" on page 24 to understand which properties and actions in IBM Content Navigator (or IBM Content Navigator for Microsoft Office) you can customize through an external data service and understand what the communication protocol looks like.

## Helpful links

See the following sources to help you get started with gaining the required skills:

- JSON tutorial describing the notation:

  http://www.w3schools.com/json/

- JSON Java API bundled with IBM Content Navigator

  http://pic.dhe.ibm.com/infocenter/p8docs/v5r2m0/topic/com.ibm.javaeuc.doc/com/ibm/json/java/package-summary.html

- External data services at IBM Knowledge Center for FileNet P8:

  http://pic.dhe.ibm.com/infocenter/p8docs/v5r2m0/topic/com.ibm.developingeuc.doc/eucap001.htm

### 1.2.3 Implementing a plug-in

An IBM Content Navigator plug-in is a powerful mechanism and the most flexible option for customizing user experience within your solution. It enables developers to add new functionality, change existing behavior and appearance of the application or create completely new application. A plug-in can implement one or more extension points to fulfill business requirements.

The following extension points are available:

- Action: Used to add a new context menu item or toolbar menu item. It is often accompanied with a Service extension point that implements the business logic for the action. For example you can add an action that will export a selected folder and its contents to a ZIP file.
- Open Action: Used to alter the behavior when opening a content item. For example, depending on a document property value, you can implement the Open Action to open different viewers or opening a specific folder display in a new feature.
- Feature: Used to add new functionality that has its own interface pane or use an existing interface pane with some modifications. You also must implement a *dijit layout* that will render the content of the feature, where you can use existing widgets or customized widgets. For example, you can implement a feature that displays a custom search form and existing content list widgets.
- Menu: Used to add configurable menu items to your widgets, instead of hard coding them in the widgets. This allows you to configure different actions for different desktops or roles from Administration Desktop. For example, you can implement a menu that displays a configured action in a newly created widget for displaying customers.
- Layout: Used to create a completely new look and feel on top of the IBM Content Navigator framework or to change the existing features in your desktop. For example, you can build a custom layout for mobile browsers that will adapt to its screen size.
- Viewer: Used to change existing viewer for a specific MIME type. For example, you can implement a viewer that can be used to view PDF documents and allow users to electronically sign the opened document.

- Request filter: Used to change the parameters (JSON request) sent from the browser to a service or to replace invoked service by sending its own JSON response. For example you can censure restricted words in a comment entered by user or restrict access to a content item by performing additional security check in external system.
- Response filter: Used to change the JSON response of an invoked service to change the behavior and appearance of the IBM Content Navigator. For example, you can determine which columns are displayed in a content list in the browse view or change the order of properties on an add document form.
- Service: Used to provide server side functionality for your action, widget or any other front-end component. You can use repository-specific API, configuration API, or your own library to invoke operations you need. For example, you can implement a service that creates folder with several subfolders in a repository.

A special type of extension point that is not represented by the Java class is *Widget*. You can use it in your plug-ins to render feature, support actions with some dialogs or create customized step processor.

A specific use case can be implemented with the *OnDemand Authentication Service* extension point that allows you to forward authentication token from IBM Content Navigator to IBM Content Manager OnDemand repository for a single sign-on (SSO) solution.

**Important:** Request and response filters change behavior and appearance of all clients using IBM Content Navigator services such as IBM Content Navigator for Microsoft Office.

A format of the JSON messages exchanged between client and server is subject to change without notice. You might need to update your code of request and response filters if newer version of IBM Content Navigator is released.

## Required skills

From a development perspective, there are two types of extensions points, each with different skills prerequisites:

- Server-side extensions: These requires that you understand Java programming language and JSON. These extension points are as follows:
  - Service
  - Request Filter
  - Response Filter
  - OnDemand AuthenticationService

- Client-side extensions: These require you to understand JavaScript programming language in addition to Java programming language and JSON. These extension points are as follows:
  - Action
  - Open action
  - Menu
  - Feature
  - Layout
  - Viewer

To develop a widget that is part of your plug-in, you must know how use the following technologies:

- Dojo 1.8.4 (This version is packaged with IBM Content Navigator)
- HyperText Markup Language version 5 (HTML 5)

IBM Content Navigator provides a Java API for implementing plug-in classes and JavaScript API for developing the client extension points. For more information, see 1.4.3, “Plug-in API” on page 27 and 1.4.4, “Content Navigator JavaScript API” on page 47.

### Helpful links

See the following sources for more information to help you get started with gaining the required skills:

- Dojo tutorial for beginners:

  http://dojotoolkit.org/documentation/tutorials/1.9/modern_dojo/

  The minimum requirement for experienced JavaScript developers is to understand the Dojo modules. How to use declare, define, and require functions as at the following web page:

  http://dojotoolkit.org/documentation/tutorials/1.8/modules/

- JSON tutorial describing the notation:

  http://www.w3schools.com/json/

## 1.2.4 Developing or integrating a custom application

Implementing new Enterprise Content Management solutions require consideration for existing applications within your organization. There might be several front-end applications that you must integrate with or there might be a unified front-end platform such as IBM WebSphere® Portal or Microsoft SharePoint through which your application will be accessed.

IBM Content Navigator provides two approaches for integrating custom applications:

- Unbound: A lightweight integration using IBM Content Navigator URL API, to render a specific desktop, feature, folder, document or search template in an iFrame of your application or in a new window. This approach has limited options for interaction with your application, but avoids cross-site scripting problems and Dojo version conflicts.
- Bound: A heavyweight integration, where all the JavaScript code is running directly within your application. This option gives you better control of the interactions with your application thought JavaScript event wiring. To accomplish this integration, you will have to initialize connection to IBM Content Navigator server and resolve any cross-site scripting problem.

To develop a custom application, you can either leverage IBM Content Navigator GUI and develop your own layout plug-in for your desktop or you can create completely new application by using IBM Content Navigator JavaScript API.

## Required skills

For unbound integration, you must understand IBM Content Navigator URL API as described in 1.4.1, “URL API” on page 23.

To develop a custom layout in IBM Content Navigator, you must understand the plug-in development as described above.

Developing a completely new application or bound integration requires the understanding of the following technologies and APIs:

- JavaScript
- JSON
- IBM Content Navigator JavaScript API
- Dojo 1.8.4 (This version is packaged with IBM Content Navigator)
- HyperText Markup Language version 5 (HTML 5)

For more details see 1.4.4, “Content Navigator JavaScript API” on page 47.

### Helpful links

See the following sources for more information to help you get started with gaining the required skills:

- Dojo tutorial for beginners:

  http://dojotoolkit.org/documentation/tutorials/1.9/modern_dojo/

- The minimum requirement for JavaScript experienced developers is to understand the dojo modules. How to use declare, define, and require functions:

  http://dojotoolkit.org/documentation/tutorials/1.8/modules/

- JSON tutorial describing the notation:

  http://www.w3schools.com/json/

- IBM Content Navigator JavaScript API reference at IBM Knowledge Center for FileNet P8:

  http://pic.dhe.ibm.com/infocenter/p8docs/v5r2m0/topic/com.ibm.developingeuc.doc/doc/JavaScriptdoc/index.html

## 1.2.5 Delivering mobile access

If you are planning to deliver access to your repository from mobile devices, one of the first considerations is whether you create a mobile application users can download or create a mobile website accessible from a browser on the phone. IBM Content Navigator framework can support both options:

- Mobile application development: Supported by the IBM Worklight® project that can be extended to your needs. It builds a hybrid application that can be deployed on several platforms such as Android and iOS.
- Web application development: Supported by IBM Content Navigator JavaScript model library, which you can use in your own application or more easily in your layout plug-in.

**Note:** IBM Content Navigator provides a mobile application for iOS devices that can be downloaded from Apple App Store. You can add new features to this application though desktop configuration, by specifying the web page (URL) of your feature.

### Required skills

Both mobile application development and web application development require an understanding of the following technologies:

- JavaScript
- HTML5
- CSS3
- Dojo Mobile

In addition, for mobile application development, you must also understand these areas:

- Application development with IBM WorkLight Studio
- Apache Cordova API

For more details see Chapter 11, “Extending solutions to mobile platform” on page 413.

### Helpful links

See the following sources for more information to help you get started with gaining the required skills:

- Extending Your Business to Mobile Devices with IBM Worklight:

  http://www.redbooks.ibm.com/abstracts/sg248117.html?Open

- IBM Worklight Version 6.0.0:

  http://pic.dhe.ibm.com/infocenter/wrklight/v6r0m0/index.jsp

- Apache Cordova:

  http://cordova.apache.org/

- Tutorial for Dojo Mobile:

  http://dojotoolkit.org/documentation/tutorials/1.8/mobile/tweetview/getting_started/

## 1.2.6 Common use cases and their development options summary

To help you with IBM Content Navigator customization and extension, this section summarizes the common use cases mentioned in this chapter, where you should look to do the customization or extension, the required skills needed to do the work, and alternatives.

The use cases are divided into the following categories:

- Adding new functionality (Table 1-1)
- Modifying existing behaviors (Table 1-2)
- Modifying existing functionality (Table 1-3 on page 16)
- Modifying appearances (Table 1-4 on page 16)
- Integrating with external applications (Table 1-5 on page 17)

*Table 1-1 Use cases for adding new functionality*

| Use case | Look for | Required skills | Alternative |
|---|---|---|---|
| Add new context menu item or toolbar menu item | Plug-in - Action and Service | Java, JavaScript | |
| Add new feature page | Plug-in - Feature and Widgets | Java, Widget library, Dojo, (Model Library) | |
| Add new document viewer for specific MIME types | Plug-in - Viewer | Java, Widget library, Dojo, Model Library | Plug-in - OpenAction |
| Implement custom step processor | Plug-in - Widgets | JSP, PE API, Dojo, Widget library, Model Library | |
| Add feature to existing iOS application | Configuration - Desktop | HTML | Mobile - Worklight |

*Table 1-2 Use cases for modifying existing behavior*

| Use case | Look for | Required skills | Alternative |
|---|---|---|---|
| Prefill property with value when creating content items, searching or working with tasks | EDS | Web application, JSON | Plug-in - Request filter |
| Alter the property to choice list or dependent choice list | EDS | Web application, JSON | Plug-in - Request filter |
| Hide properties when editing content items or working with task | EDS | Web application, JSON | Plug-in - Request filter |
| Make properties non editable when editing objects or working with task | EDS | Web application, JSON | Plug-in - Request filter |

| Use case | Look for | Required skills | Alternative |
|---|---|---|---|
| Implement custom sorting mechanism for search results | Plug-in - Response filter | Java | |
| Restrict access to content item or to execute a service | Plug-in - Request filter | Java | |

*Table 1-3 Use cases for modifying existing functionality*

| Use case | Look for | Required skills | Alternative |
|---|---|---|---|
| Add front-end validations for properties when editing content items or working with task | EDS | Web application, JSON | Plug-in - Response filter |
| Alter existing feature | Plug-in - Layout and Widgets | Java, Widget library, Dojo | Plug-in - Feature |
| Change the existing open action for a content item | Plug-in - OpenAction | Java, JavaScript | Plug-in - Viewer |
| Censure comments entered by user | Plug-in - Request filter | Java | |
| Implement SSO solution for CMOD | Plug-in - OnDemand Authentication Service | Java | |

*Table 1-4 Use cases for modifying appearances*

| Use case | Look for | Required skills | Alternative |
|---|---|---|---|
| Change the logos and banner colors | Configuration - Desktop | none | Plug-in - Layout |
| Change the columns in content list when browsing in general | Configuration - Repository and Desktop | none | Plug-in - Response filter or Feature and widget |
| Change or add columns in content list for specific folder or at specific condition | Plug-in - Response filter | Java | |

| Use case | Look for | Required skills | Alternative |
|---|---|---|---|
| Change the property formatter for a cell in the content list | Plug-in - Response filter | Java, JavaScript | |
| Change which properties are available for user in search or system properties widgets | Configuration - Repository and Desktop | none | Plug-in - Feature and widget |
| Change the look and feel of whole desktop (such as for mobile browser) | Plug-in - Layout and Widgets | Java, Widget library, Dojo, CSS | Custom app |
| Change the order of properties in add/edit/view properties widgets | Plug-in - Response filter | Java | Entry template |
| Change the property formatter or property editor of properties in add/edit/view properties widgets | Plug-in - Response filter | Java, JavaScript | |

*Table 1-5 Use cases to integrate with external applications*

| Use case | Look for | Required skills | Alternative |
|---|---|---|---|
| Integrate feature, search template or folder list into own application | URL API | none | Custom app |
| Display ICN widget in own application | Custom app | Dojo, Widget library, Model library | iFrame pointing to desktop with own layout or to own feature |
| Deliver mobile application | Mobile - Worklight | Worklight studio, Cordova API | Dojo, Model library |
| Deliver web application accessible from mobile | Plugin - Layout and Widgets | Java, Widget library, Dojo mobile | Custom app |

# 1.3 IBM Content Navigator development architecture

The IBM Content Navigator architecture is based on modularity and clear separation of presentation layer, data access layer, and midtier services. The modularity allows you to add new functionality without affecting existing functionality through plug-in mechanism; the separation into layers provides an option to replace a specific layer without reworking the entire application.

Figure 1-1 shows the virtual layers of IBM Content Navigator and depicts the dependencies of the components.

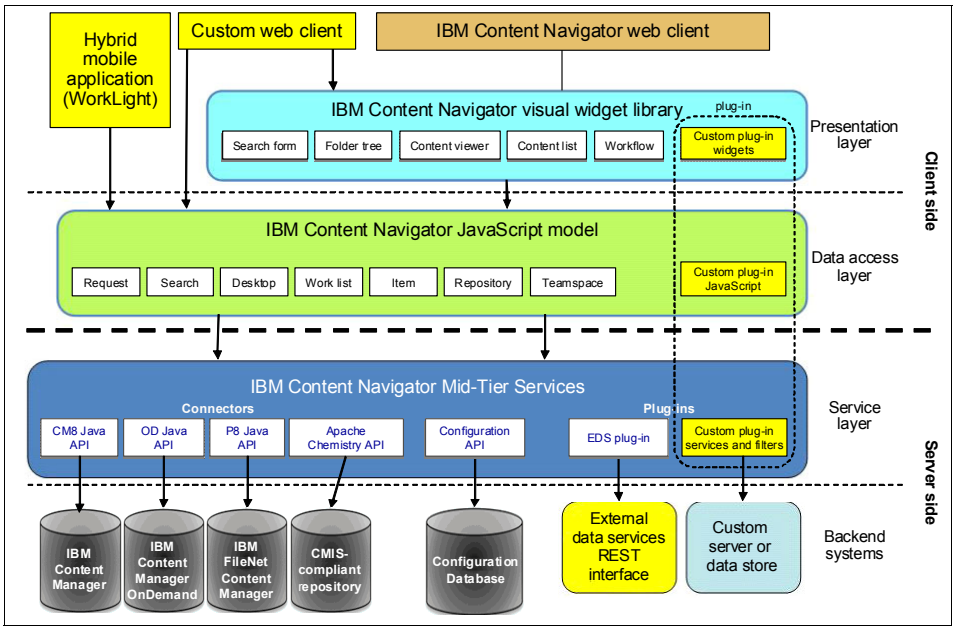



*Figure 1-1 IBM Content Navigator development architecture*

Presentation layer as the top-most level has two main responsibilities: it renders the output to the user and handles interaction with user; it uses data access layer to store, retrieve and update the data model of the application. The main component of the presentation layer is *IBM Content Navigator visual widget library*, which contains a set of visual widgets extracted from any business logic. It is built on Dojo 1.8.4 and IBM Dojo Extension for OneUI (IDX) 1.3.0.4 and extends the Dojo, Dojo widgets, and IDX libraries. The visual widgets such as content list, folder tree, dialogs, search form, and others, builds IBM Content Navigator web application user interface, but can also be used in your custom web client.

The data access layer is responsible for holding the application data and for providing transparent access to the repositories and services running on the server. It also allows you to attach an event listeners for any data model changes. The main component, IBM Content Navigator JavaScript model, abstracts repository-specific functions into a common interface and offers unified (repository-independent) approach to retrieve, create, delete or modify content in the underlying repository. The data model optimizes the midtier access through caching mechanisms and smart requests. It is implemented as JavaScript library that can be easily included and used in your own applications.

The service layer is responsible to provide a complete set of services for your Enterprise Content Management solution. It is designed as an integration layer between the front-end application and the back-end systems. IBM Content Navigator Midtier services is a server component implemented in Java and provides services and interfaces to all supported repositories and to IBM Content Navigator configuration database. It contains a plug-in mechanism that allows you to augment or modify functionality at each layer. External data service plug-in, which is part of IBM Content Navigator, exposes a REST interface that you can use to integrate your external system into IBM Content Navigator for specific use cases without the need to implement a plug-in.

### 1.3.1 Programming interfaces

Each layer from IBM Content Navigator architecture provides a programming interface that you use to deliver a custom user experience. Depending on the layer you want to change, you can use these interfaces:

- URL API: Used at the presentation layer to integrate IBM Content Navigator into custom web application.
- External data services (EDS): Defines the REST interface that allows you to change the presentation layer, which is the behavior when editing properties of documents, search templates, folders and step processors.
- Java API:
  - Plug-in API: Defines a Java interface that allows you augment or modify each layer of IBM Content Navigator architecture. It contains set of abstract and utility classes for building your own plug-ins.
  - Configuration API: Used by midtier services layer to access or update configuration of IBM Content Navigator and user configuration.

- JavaScript API:
  - Modeling: Used by presentation layer to access the data from the server. It contains a set of JavaScript classes that custom web application or Worklight application can use to connect to midtier services.
  - Widgets: Used at presentation layer to create new or extend existing widget catalog of IBM Content Navigator.

See the following reference information in the IBM Knowledge Center:

- A complete Java API reference:

  http://pic.dhe.ibm.com/infocenter/p8docs/v5r2m0/topic/com.ibm.javaeuc.doc/overview-summary.html

- A complete JavaScript API reference:

  http://pic.dhe.ibm.com/infocenter/p8docs/v5r2m0/topic/com.ibm.developingeuc.doc/doc/JavaScriptdoc/index.html

## 1.3.2 Communication flows

To understand IBM Content Navigator architecture, you must understand the flow of the information exchanged between layers. There are two primary communication flows in IBM Content Navigator architecture:

- Communication between widget library and JavaScript model
- Communication from JavaScript model to midtier services

### Widget to model communication

Although the communication between widget library and JavaScript model is bidirectional, there is no dependency in the JavaScript model on the widget library. The bidirectional communication is achieved by an event-driven approach. The model library publishes several events that defines the change in the object's state. An example might be that a desktop was loaded or a document property was changed. A visual widget can subscribe a listener to the event (in Dojo terminology, this is "connect a function to the event"). This listener or function is then automatically called when the event occurs. The same approach is used in widget-to-widget communication when you need to connect two widgets together.

Figure 1-2 shows communication flow during desktop initialization. The LoginPane registers for an onDesktopLoaded event, and the NavigatorMainLayout registers for an onLogin event during the postCreate phase of the widget initialization.

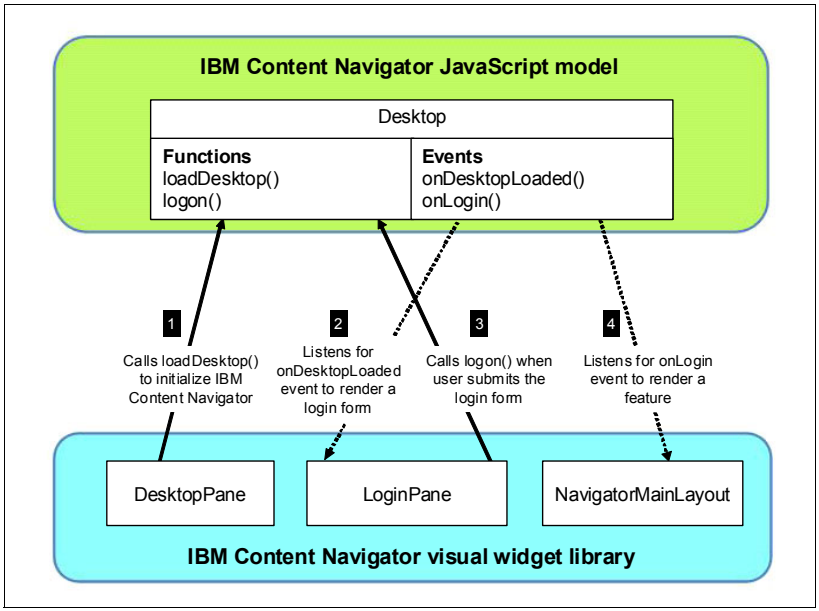



*Figure 1-2 Communication flow between widget library and JavaScript model*

## Model to midtier services communication

All communication from JavaScript model library to IBM Content Navigator midtier services is done through ecm.model.Request JavaScript class. It provides a common way to handle session, timeouts and errors. Generated service call sends HTTP POST request containing parameters encoded in JSON format.

The server side uses a controller to handle incoming service calls. The controller distributes the service call to a specific class as defined in the `struts-config.xml` file. The plug-in mechanism of IBM Content Navigator allows you to implement request and response filters that intercept the original flow. The request filter allows you to modify incoming JSON message before the requested action is invoked. It also allows you to return your own response and skip any additional processing. The response filter is executed after the requested service is invoked. It allows you to modify JSON response returned by the service.

Figure 1-3 show communication flow.

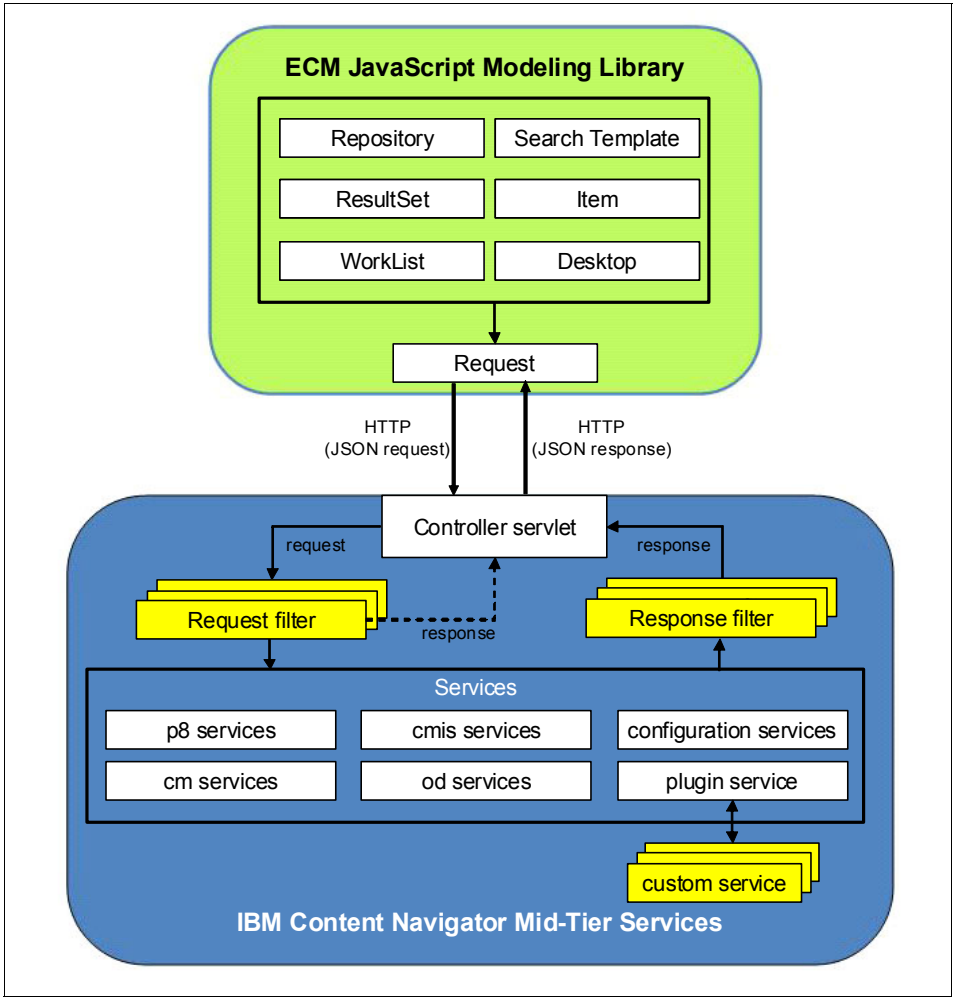



*Figure 1-3 Communication flow from model library to midtier services*

# 1.4 Developing with IBM Content Navigator APIs

IBM Content Navigator APIs consist of a URL API, external data services, plug-in API, and Content Navigator JavaScript API.

## 1.4.1 URL API

The easiest way to integrate a document management solution into your application is through a URL link. IBM Content Navigator allows you to access desktops, features, documents, search templates, and folders directly by embedding constructed URL link into your application. You can use the following URL types:

- Desktop URL: Opens a specified desktop in the full IBM Content Navigator client.
- Feature URL: Opens a desktop with the specified feature preselected, ignoring the default feature setting for the desktop.
- Folder URL: Opens a view that displays the sub-tree and content of specified folder.
- Document URL: Opens the document in the preconfigured viewer for the MIME type of the document. If the document is a StoredSearch, it will open the search form.

For integration through iFrame, you can also render a particular feature or desktop without the context of the entire desktop. You can hide the banner across the top of the window or hide the Features sidebar by appending the *sideChrome* parameter to the URL.

As an enhancement for the URL API, you can also implement a custom layout plug-in or a custom feature, which renders widgets you want to display in your application or portal. A good approach is to combine both the layout and the feature for your integration scenario. Create an “integration” desktop with your custom layout, containing all custom features you want to integrate.

An important consideration when integrating application through the URL is a single sign-on solution between your application and IBM Content Navigator, so the user does not need to log in to IBM Content Navigator when redirected from your application or when part of IBM Content Navigator is rendered within your application.

More details about construction of the URL and available parameters are in the following locations:

- IBM FileNet P8:

  http://pic.dhe.ibm.com/infocenter/p8docs/v5r2m0/topic/com.ibm.developingeuc.doc/eucbd001.htm

- IBM Content Manager:

  http://pic.dhe.ibm.com/infocenter/cmgmt/v8r4m0/topic/com.ibm.developingeuc.doc/eucbd001.htm

- IBM Content Manager OnDemand:

  http://pic.dhe.ibm.com/infocenter/cmod/v9r0m0/topic/com.ibm.developingeuc.doc/eucbd001.htm

### 1.4.2 External data services

The external data services (EDS) is an interface that IBM Content Navigator provides to access data from external data sources. EDS can be used to define field validation criteria, prefill the field values, or other custom properties for existing repository metadata during delivery or change of documents or folders.

EDS provides the following options to change the behavior of properties when adding or changing document or folders, based on the document class. If a document class has the same name in IBM Content Manager and IBM FileNet Content Manager, the following defined rules are valid for both document classes:

- Create a choice list for attributes based on a data source lookup.
- Prefill attributes with default values.
- Create choice list dependencies. For example, if a user must enter the names of the state and the city, EDS can provide two choice lists: one that contains the states, and one that contains the cities. If a user selects a state, then only the cities that are located in this state are displayed in the second choice list. The second choice list is therefore dynamically loaded by EDS, depending on the first selection in the state-list.
- Define a maximal and a minimal length of properties.
- Set properties to be hidden, required, or read only.
- Define regular expressions for the validation of user input.

Using EDS ensures a high level of valid metadata because it avoids typographical errors during the delivery of new items. Using EDS also improves the search results. After EDS is enabled for property validation or choice list, the defined rules are applied to every IBM Content Navigator form using the defined document classes and their properties.

A choice list can be created from any existing data source or repository that IBM Content Navigator can access. The choice list values can be obtained from a simple text file or through complex database tables. The use of a JSON file is also possible.

Technically, EDS is a plug-in that is provided by IBM Content Navigator. The customer-specific implementation of EDS and the connection to the various repositories or validation files is done in a web application that must be implemented and deployed. Next, the URL to the custom EDS implementation must be configured in the EDS plug-in configuration.

Figure 1-4 on page 26 shows the general workflow of EDS. The process consists of the IBM Content Navigator client, which is the user front end, the IBM Content Navigator server, which holds the EDS plug-in itself, and its configuration to the EDS web application. The communication between IBM Content Navigator and the EDS Service is based on the REST protocol.

The web application accesses the external data source, such as a JSON file, external database, or a simple CSV file.

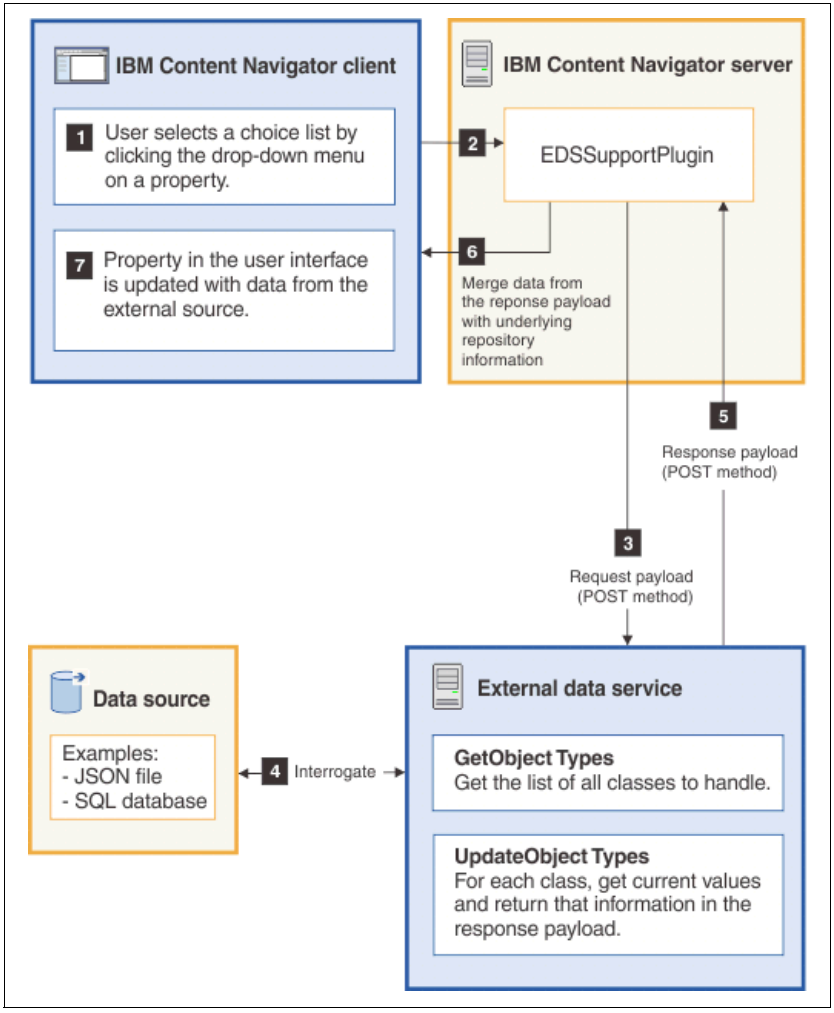



*Figure 1-4 External data services flow of information*

If a user selects a choice list for a property that is provided and enabled by EDS, the IBM Content Navigator Server EDS plug-in is called and forwards the request to the configured EDS web application as a POST request.

The requestMode provides a set of values, depending on the executed action in the front end:

- initialNewObject: Delivered if a new object is being created.
- initialExistingObject: Delivered if an existing object should be modified.
- inProgressChanges: Delivered if an existing object should be modified and the dependent choice list is defined for a property.
- finalNewObject: Delivered before the new object is persisted in the datastore, such as check in or final add.
- finalExistingObject: Delivered before the existing object is persisted, for example by selecting save action.

The servlet that implements the POST request accesses the data source that contains the validation rules and delivers the response to the IBM Content Navigator server EDS plug-in. Next, the plug-in merges the rules from the external data source with the repository-specific rules, and displays the result in the front end.

Implementing a new EDS can be summarized with the following steps:

1. Create a web application that contains the implementation of doGet() and doPost() requests, and deploy the web application.
2. Deploy the EDS plug-in that is delivered with IBM Content Navigator and configure the plug-in to use the web application URL that was created and deployed in step 1.

### 1.4.3 Plug-in API

IBM Content Navigator plug-ins are powerful mechanisms that provide extensive capabilities for extending the product to meet your specific business needs. Through the use of plug-ins, you can add functionality directly into the IBM Content Navigator user interface, or you can completely customize the user interface to meet your business needs.

To implement a plug-in, you must implement a set of abstract Java classes that provide IBM Content Navigator with information about which functionality is provided by the plug-in and how that functionality should be integrated with the base product. Additionally, you might need to implement JavaScript classes depending on what type of extension points your plug-in contains.

Figure 1-5 and Figure 1-6 show the plug-in extension point classes and the dependencies between them. Blue links in the diagrams express indirect dependency specified by identificator or file name of the object instead of real object reference.

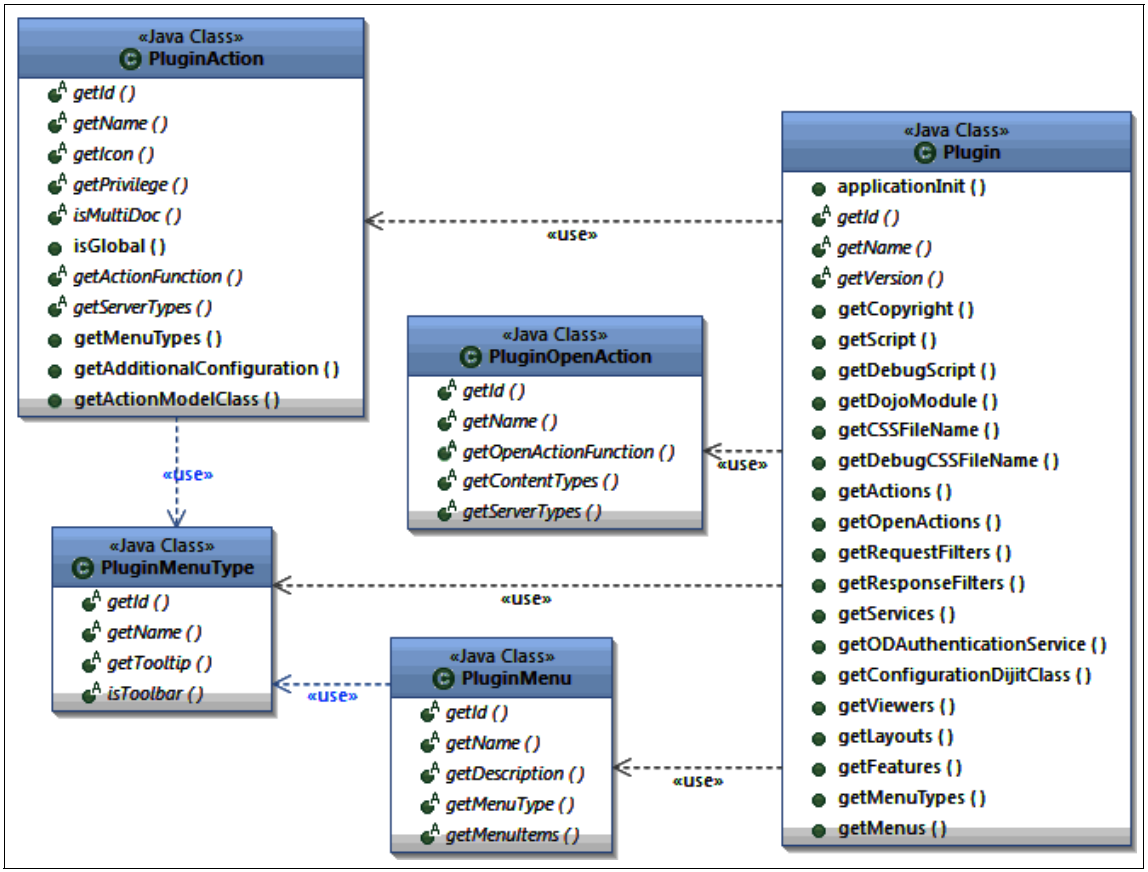



*Figure 1-5 IBM Content Navigator Plugin extension points (part 1 of 2)*

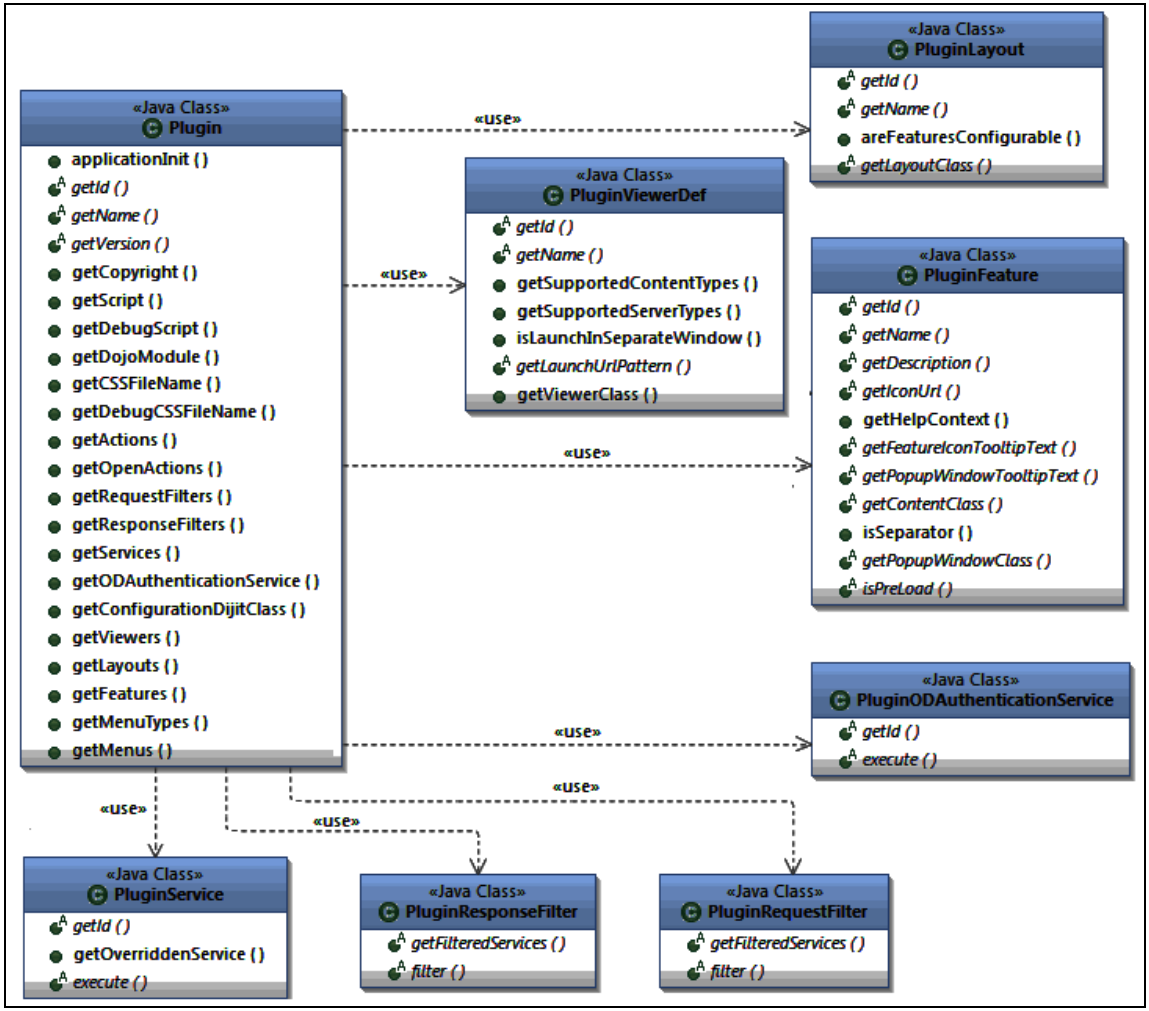



*Figure 1-6 IBM Content Navigator Plugin extension points (part 2 of 2)*

## Plug-in class

Your plug-in must provide an implementation of the abstract Plugin class. It has basic information for the IBM Content Navigator plug-in container about the name, ID and version of the plug-in and which extension points are implemented within this plug-in. An overriding method in the Plugin class can specify additional resources that are used by the plug-in. These resources are as follows:

- Configuration of dijit class
- Script file and debug version of the script file
- Cascading Style Sheets (CSS) file and a debug version of CSS file

**Note:** All resources used by the plug-in must be in the `WebContent` subfolder created in the Java package folder of the plug-in class. See Figure 3-5 on page 83.

The `WebContent` folder represents the root folder for your resources and Dojo packages. Any reference to your resource file or widget must be relative from the `WebContent` folder.

**Important:** A method getDojoModule() must be implemented to allow using dojo.require with path names that are relative to the `WebContent` folder.

Figure 1-7 shows plug-in and its resources directly defined by the Plugin class.

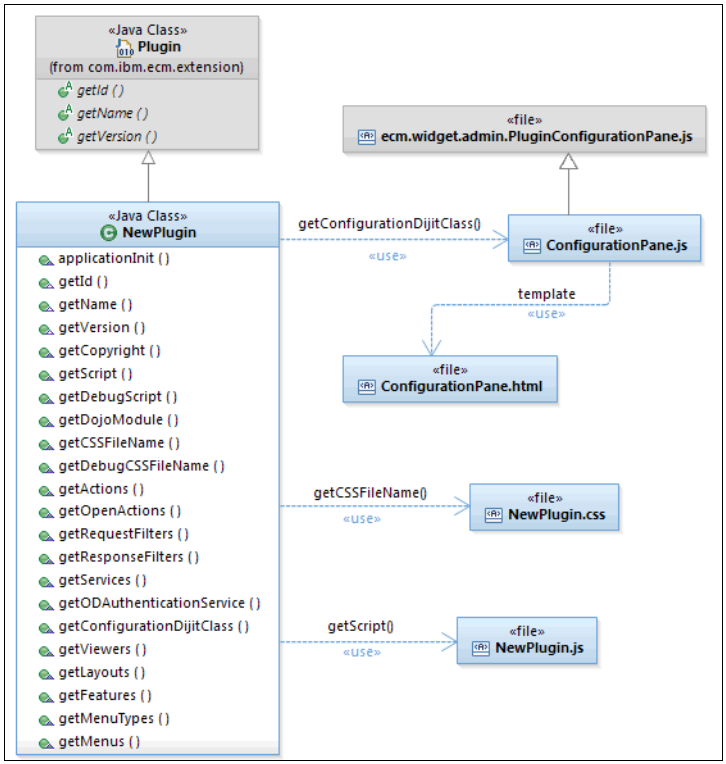



*Figure 1-7 Class diagram for custom Plugin class*

Configuration dijit class is specified when your plug-in requires an ability to configure the plug-in through Administration Desktop without doing any coding (for example, to configure a server host name of your external system used in plug-in service for the specific environment). The specified dijit class must extend `ecm/widget/admin/PluginConfigurationPane` dijit, where you can define a user

interface for your configuration parameters. If you specify the configurationString attribute and call the onSaveNeeded(true) method in your dijit, the framework will manage the save process automatically for you. To initialize the user interface with stored configuration values you must override the *load* method and provide code that will extract your configuration values from configurationString. For a complex set of configuration parameters, you can encode your parameters and values in configurationString into JSON format.

A script file or debug script file is required when your plug-in needs to execute some JavaScript code during initialization of IBM Content Navigator, for example to instantiate custom widgets used in your plug-in. The script is also required for definition of global functions used by your OpenActions plug-in. The script is executed during the application initialization after the `Desktop.loadDesktop` call and before onDesktopLoaded event is called. If debug script file is not specified, IBM Content Navigator will use the script file when running in debug mode.

The Cascading Style Sheets (CSS) file defines the rules for rendering the HTML elements. You can define custom rules used by your widgets. You can also override existing rules of IBM Content Navigator, but it is not recommended because the Content Navigator rules are not exposed as API, and may change from one release to another. However, the CSS file from the latest defined plug-in has the highest priority when the style for HTML element is calculated.

**Note:** CSS is loaded asynchronously and may load after Dojo has initialized and sized layouts. This means that the provided CSS should not add rules that can affect the layout (such as widths, heights, visibility) because these rules might not be considered on initial layout.

Implementing a new plug-in can be summarized with the following steps:

1. Create a new Java class that extends the com.ibm.ecm.extension.Plugin class.
2. Optionally add configuration options for the plug-in:
   a. Create a new dijit that extends, at least, the following dijit:

      `ecm.widget.admin.PluginConfigurationPane`

   b. Implement the getConfigurationDijitClass() method in the Plugin Java class to return the name of the configuration dijit.
   c. Implement getDojoModule() method in the Plugin Java class, to return the package of your digit.

3. Optional: Add custom widgets and JavaScripts:
    a. Create a new widget with all required elements (for example, template file and JavaScript file).
    b. Create a JavaScript file that instantiates the new widget in the plug-in.
    c. Implement the getScript() method in the Plugin Java class to return the name of the JavaScript file from previous step.
    d. Implement getDojoModule() method in the Plugin Java class, to return the package of your widgets.
4. Optional: Add a custom CSS file:
    a. Create a new CSS file.
    b. Implement the getCSSFileName() method in the Plugin Java class, to return the name of the CSS file.

## Action

A menu action extends current IBM Content Navigator functionality by providing a mechanism to insert a new context menu item or toolbar menu item. The action can be added to existing menus or it can be added to a new menu. When you implement the action, you also implement the functionality that is provided by the new action, which is typically done by adding a new service. An example of an action is to update a hidden attribute of a document or folder. If the new action is available to users through one of the existing IBM Content Navigator menus, for instance the ContentList menu, this menu must be copied and updated with the new action in the IBM Content Navigator administration console.

When you define the action, you define several characteristics about the action, including the following characteristics:

- Whether the action is available in a toolbar or a menu
- Whether the action is global (displayed directly below the banner)
- Which privileges of the item must the user have in order to invoke this action
- Whether the action can be invoked when multiple items are selected
- Which type of repositories the action applies to
- Which classes or items in a content list the action can be invoked for

An action consists of a Java class that defines the action parameters, and a JavaScript model class that implements the action.

Figure 1-8 shows an overview of components that must be involved by an action.

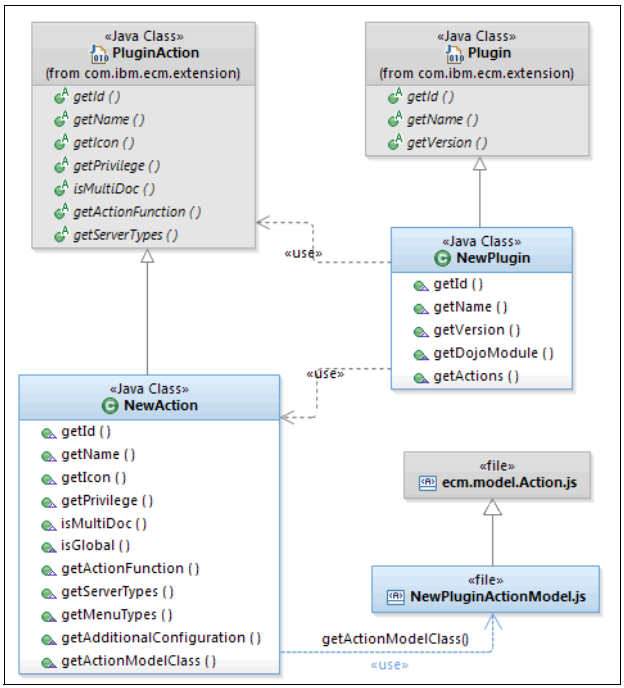



*Figure 1-8 Class diagram for an action*

A Java class extends the PluginAction class and provides the basic parameters, such as name and privileges for the new action. The JavaScript model class supplements the Java class and implements the behavior of the action. You can override the isEnabled function of the model class to hide the action for specific classes or items fulfilling your custom condition.

Implementing a new action can be summarized with the following steps:

1. Create a new Java class to extend the `com.ibm.ecm.extension.PluginAction` class, and provide details for the new action.
2. Create a new Dojo class that extends the `ecm.model.Action` class and override the performAction function.
3. Implement the getActionModelClass() method in the Action Java class to return the name of the Dojo class created previous step. Alternatively, you can implement getActionFunction() that returns the name of the global function defined in `plugin.js` file, but the getActionModelClass is the preferred way.
4. Implement the getActions() and getDojoModule() methods in the Plugin Java class.

**Note:** Make sure the getActionFunction() does not return null. It must return at least an empty string if you override performAction in the ActionModel javascript class. Otherwise, although your plug-in will be loaded correctly, you will not be able to add the action into a menu (Content Navigator will not show your action in a list of available actions).

## OpenAction

An OpenAction is a built-in action to open documents or folders, for example by double-clicking an item in the tree or in the content list. By extending the PluginOpenAction abstract class, you can change the behavior of existing OpenAction. Instead of opening a viewer defined for specific MIME type you can for example first open a dialog prompting to accept terms and conditions for accessing this content and then open a viewer.

When you define the OpenAction, you must define MIME types and server types for which specified function will be invoked.

Figure 1-9 shows an overview of the components that must be involved by an OpenAction.

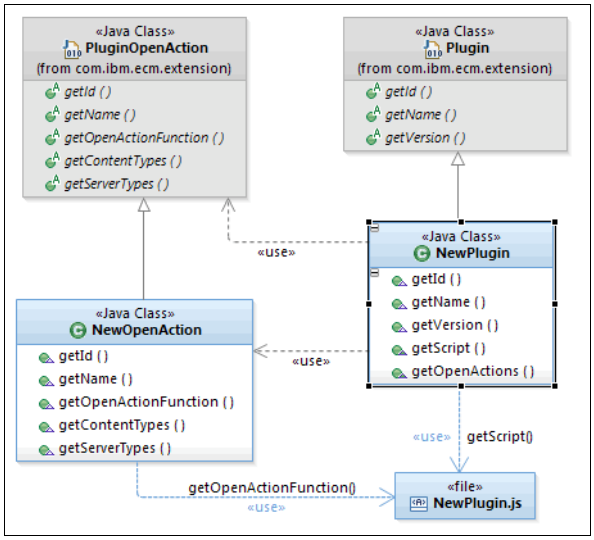



*Figure 1-9 Class diagram for an OpenAction*

A NewOpenAction extends the PluginOpenAction class and provides the information about supported MIME types and server types. The JavaScript function specified in getOpenActionFunction method has to be defined globally and therefore implemented in plug-in's JavaScript file.

Implementing a new OpenAction can be summarized with the following steps:

1. Create a new Java class that extends the `com.ibm.ecm.extension.PluginOpenAction` class, and provide details for the new action.
2. Define and implement global JavaScript function with your own logic for OpenAction in plug-in's script file.
3. Implement the getOpenActions() method in the Plugin Java class.

**Note:** Viewer definition for specified content type has a higher priority than OpenAction. Therefore, for your OpenAction to work, you must remove the specified content type from the viewer map definition for your desktop.

## Menu and MenuType

IBM Content Navigator has menus that are associated with various widgets in the user interface. An administrator can use the Administration Desktop to modify menus to add actions or remove actions from these menus. This task is done by copying the default menu for a specific menu type, modifying the copied menu, and then associating the modified menu with a widget in the user interface that uses that menu type.

In a plug-in, you can take this concept even further by defining new menu types and new actions that can be associated with the menus created for these menu types. These new menu types can then be used by your custom widgets to provide the functionality that you need. Although you do not need to implement MenuType for your widget, implementing it allows an administrator to customize the menu. You can create multiple menus associated with the menu type and then assign a specific one to the widget, based on the current user or the type of object currently associated with the widget.

When you define a new menu, you make decisions about items and menus:

- Which items (actions) should be in the menu?
- Is the menu a context menu or a toolbar?

A menu typically consists of a Java class that defines the menu properties, the actions that should be included, and the type of the menu. The menu items of a menu are actions, which are defined as described in "Action" on page 32. A menu can be used in a custom widget by using `ecm.widget.ActionMenu` widget. Use the `desktop.loadMenuActions` function to load the actions for your menu.

Figure 1-10 shows an overview of the components that are involved in a menu and menu type.

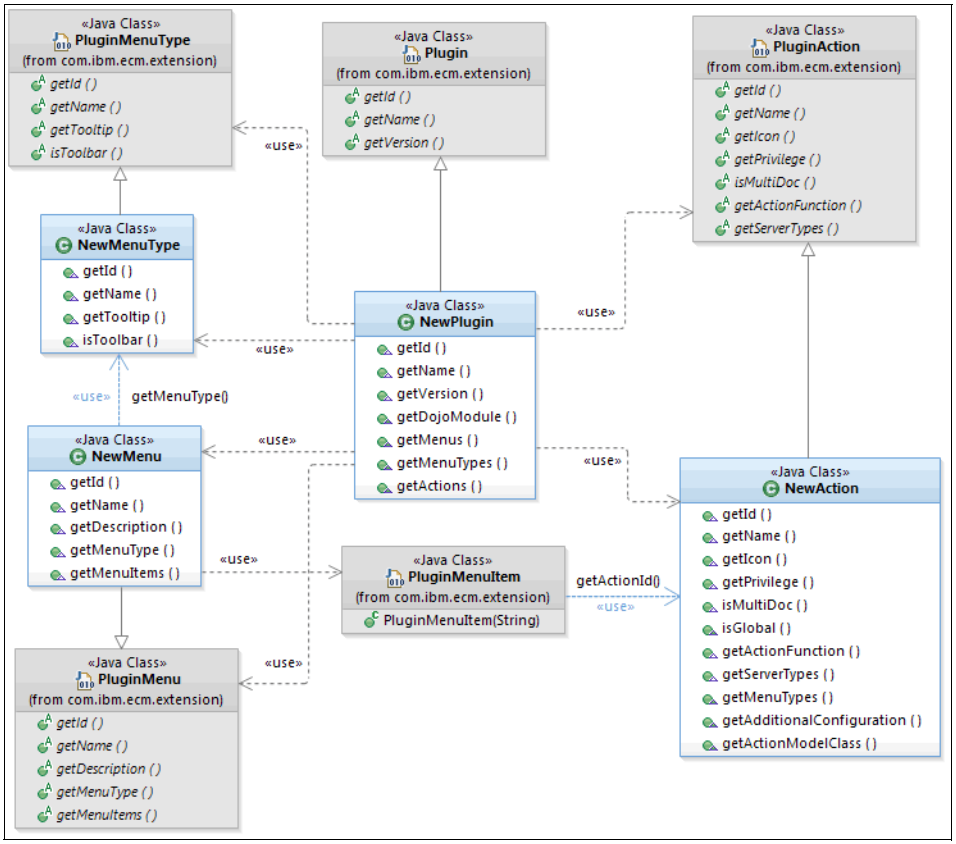



*Figure 1-10 Class diagram for a menu and menu type*

A menu consists of various properties that must be defined through several classes. The first one is a class that extends the PluginMenuType class to define whether the menu is a context or toolbar menu. The type of menu, referenced by the MenuType ID, is assigned to the new Menu class, which extends the PluginMenu class. The new Menu class can also contain a list of menu items that are listed in the menu. A menu item is typically an action that shows up in the menu. The PluginMenuItem class can be instantiated directly, without the need to extend it. An ID of the action must be specified when creating instance of the PluginMenuItem class.

Implementing a new menu can be summarized with the following steps:

1. Create a new Java class that extends the `com.ibm.ecm.extension.PluginMenuType` class, and provide details for the menu type (for example, if it is a toolbar menu or a context menu).
2. Create a new Java class to extend the `com.ibm.ecm.extension.PluginMenu` class. For the class, provide details for the menu (for example, reference the menu type created in step 1) and define menu items for the new menu (for example, an ID of an action created earlier or an existing action in IBM Content Navigator).
3. Implement the getMenus() and getMenuTypes() methods in the Plugin Java class.

## Feature

IBM Content Navigator provides a Feature pane that displays functions that are available for a desktop. For instance, the base Feature pane includes Favorites, Browse, Search, Work, and Teamspaces. In the Administrator desktop, the administrator feature is added and you can decide which parts of the feature should be available to the user. When you select a feature from the Feature pane, the associated view is displayed in the client.

You can create a plug-in to add a custom feature to the client. This feature can then provide a custom view with associated menus and actions that can be used for a particular business process. For instance, you can create a dashboard feature that displays information about the current state of policy applications. This feature will display custom widgets that displayed the number of applications received, processed and approved. It might also provide a custom action such as creating a snapshot and sending it to specific users. The new feature component can be assigned to a specific desktop through the Administration Desktop of IBM Content Navigator.

For a new feature component, you can decide about icons and widgets:

- Which icons should represent the feature?
- Which widget should be shown if the new feature is selected?

Figure 1-11 shows the components of a feature.

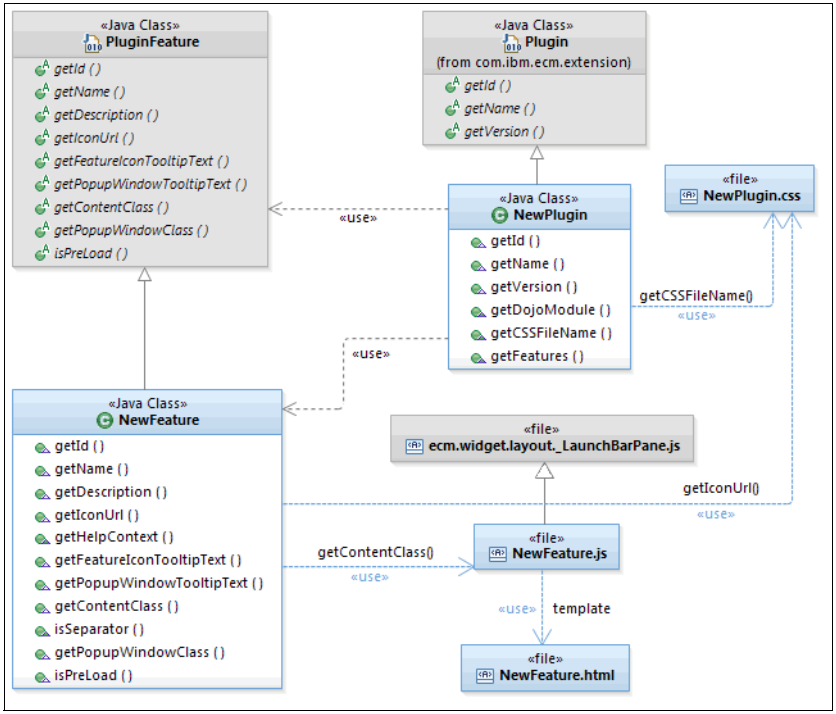



*Figure 1-11 Class diagram for a Feature*

To create a new feature, extend the PluginFeature class and define an icon, which appears in the Feature pane. Also define a content class that specifies the widget to be displayed when the feature is selected. If only one of these methods is implemented, the new feature will not be displayed. The icon must be 32 x 32 pixels image. The widget, which must be returned in the getContentClass() method, must extend _LaunchBarPane digit layout.

To define the icon that represents the new feature in the Feature pane, the getIconURL() method must return a CSS class name defined in CSS file of the plug-in. See step 4 on page 32 for details.

Implementing a new feature can be summarized with the following steps:

1. Create a new icon that represents the feature in the web client. Define new CSS class in your CSS file with background-image rule pointing to your icon.
2. Create a new widget that extends `ecm.widget.layout._LaunchBarPane` class to provide the content pane for the feature.

3. Create a new Feature Java class by extending the abstract `com.ibm.ecm.extension.PluginFeature` class, and then define the icon CSS class name in getIconURL method and the name of the Content class widget in getContentClass method created in previous steps.
4. Enhance the CSS plug-in file to define the path to the icon file.
5. Implement the getDojoModule() and getFeatures() methods in the Plugin Java class

## Plug-in service

Some plug-ins provide a customized user interface to perform only a specific task; however, most plug-ins add new functionality that extend the capabilities of IBM Content Navigator and need to be run on the server. IBM Content Navigator provides the capability to implement new services within the plug-in architecture. These new services can then be called by your custom actions or widgets.

To invoke your service from the JavaScript code you can use this function:

`ecm.model.Request.invokePluginService()`

It requires you to specify the ID of the plug-in and ID of the service. Optionally you can provide request parameters for your service and a callback functions when request completes or fails. In special cases, you might need to access your services by URL. You can construct the URL as shown in Example 1-1, but you must append the security token by calling the `Request.appendSecurityToken(url)` method.

*Example 1-1 URL pattern to access plug-in service*

```
http://<server>:<port>/<context_root>/plugin.do?plugin=<plugin_id>&acti
on=<service_id>&<custom_service_parameters>
```

The services are secured by default and cannot be called by an unauthenticated user, for example if the user's session expires, or before the user logs in, in the plug-in JavaScript file. You can allow execution of your service before the user logs in, by overriding the isSecureService method so it will return false.

Figure 1-12 shows components of a service that are typically used.

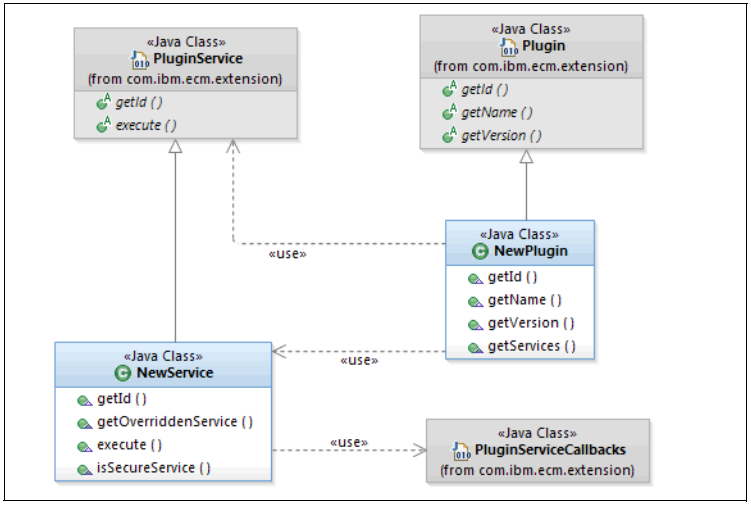



*Figure 1-12 Class diagram for a Service*

The NewService class extends the abstract PluginService class and implements the basic methods and the behavior of the new service. The execute method contains the business logic of your service. The method is supported by the PluginServiceCallback class that is included in parameters of the method together with HTTP request and response objects. The PluginServiceCallback provides connections to the underlying repositories, access to configuration database and a set of methods for obtaining the items from various repositories.

**Note:** The getOverridenService() method is no longer used. To override an existing service, use the RequestFilter that will return the response to the client.

Implementing a new plug-in service can be summarized with the following steps:

1. Create a new class that extends the PluginService class and implement the execute() method to define the behavior of the service.
2. Implement the getServices() method in the Plugin Java class.
3. Optional: If the service is activated through a custom action, create a new Action, and call `Request.invokePluginService()` in the performAction function of the JavaScript ActionModel class.

## Viewers

IBM Content Navigator provides an interface to define new viewers, which can be enabled for specific document types (depending on the MIME type of the document) and repositories. This interface provides an easy way of customizing IBM Content Navigator with well-known viewers.

When you define the viewer, you define several characteristics about the viewer, including the following characteristics:

- Whether you use a default viewer widget or a custom viewer widget.
- Whether the viewer should be opened in a separate window.
- What the supported repositories are.
- What the supported content types are.

Choosing the default viewer widget (`ecm.widget.viewer.IframeDocViewer`) requires you to specify the URL for the iFrame by implementing the getLaunchURLPattern() method. The returned URL specifies the address of the HTML document embedded in the iFrame of standard IframeDocViewer widget. The URL can point to external system to obtain content of the iFrame or can also point to the plug-in's service that returns the content you want to render for the user.

A more flexible approach is to implement custom viewer widget and override `getViewerClass()` method. Specified viewer widget class has to extend from `DocViewer` class and override the `showItem()` function. Usually the viewer widget uses custom service to obtain the content of the document or other document related information you want to render for the user.

Figure 1-13 on page 42 shows the components that are usually implemented to define a new viewer.

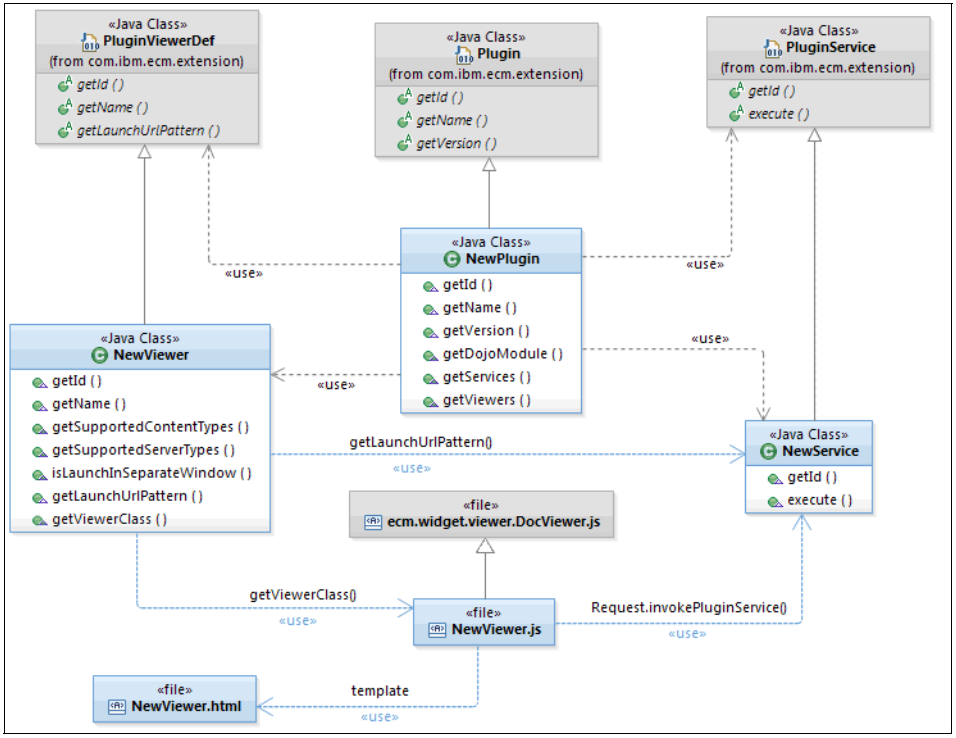



*Figure 1-13 Class diagram for a viewer*

A plug-in that contains a new viewer consists of a class that extends the PluginViewerDef to define the ID, name, and supported content and server types of the viewer. Additionally, you must either override the getViewerClass() method for custom viewer widget implementation or implement getLaunchUrlPattern() method if using the default IframeDocViewer. The returned string from getLaunchUrlPattern() is evaluated as a JavaScript expression during run time and allows you to use the following variables in the expression:

- servicesUrl: The URL to IBM Content Navigator
- docId: The identifier for the document
- docName: The name of the document. The name can be displayed in the user interface
- mimeType: The MIME content type of the document
- serverType: The content repository type (cm, od, p8, cmis)
- docUrl: A URL that will load the document content
- privs: A string of privileges for the document

For FileNet repository you can also use these variables:

- vsId: For IBM FileNet P8 documents
- replicationGroup: For IBM FileNet P8 documents
- objectStoreName: For IBM FileNet P8 documents

Alternatively you can use the entire object “item,” which is an instance of ContentItem class for the document, but it can be subject to change in the next release.

Example 1-2 shows the return value of this method.

*Example 1-2 Sample implementation of getLaunchURL method*

```
return "'http://localhost/convertDocument?documentId='+docId+'&docUrl='
+encodeURIComponent(docUrl)+'&contentType='+mimeType+'&targetContentTyp
e=application/pdf'+privs";
or
return "servicesUrl+'/plugin.do?plugin=NewPlugin&action=NewService&docu
mentId='+docId+'&docUrl='+encodeURIComponent(docUrl)+'&contentType='+mi
meType+'&targetContentType=application/pdf'+privs";
```

Implementing a new viewer can be summarized with the following steps:

1. Create a new class that extends the PluginViewerDef class, and implement either the getLauchURLPattern() method or the getViewerClass().
2. If a custom viewer widget is used, then do these steps:
   a. Create a new widget that extends `ecm.widget.viewer.DocViewer` class to render the content of the document.
   b. Implement the getViewers() and getDojoModule() methods in the Plugin Java class.
3. If the default IframeDocViewer widget is used, then do these steps:
   a. Implement the getViewers() method in the Plugin Java class.
   b. Make sure the provided URL is accessible from the clients.
4. Create a new Plugin class that extends the Plugin class of IBM Content Navigator, and instantiate the new ViewerDef class that was created in step 1 and the new PluginService that was created in step 2. If you decide to create an action to open the viewer, also define the new action in the Plugin class.

## Layout

Although widgets provide individual user interface components, you create a layout if your custom widgets provide a full user interface in the IBM Content Navigator desktop. When you create a layout, specify the widget that provides the main user interface, where you can display features and other user interface components of IBM Content Navigator in a different arrangement. Also provide a PluginLayout class that points to the widget and defines the name of the new layout. Figure 1-14 shows the components of a layout.

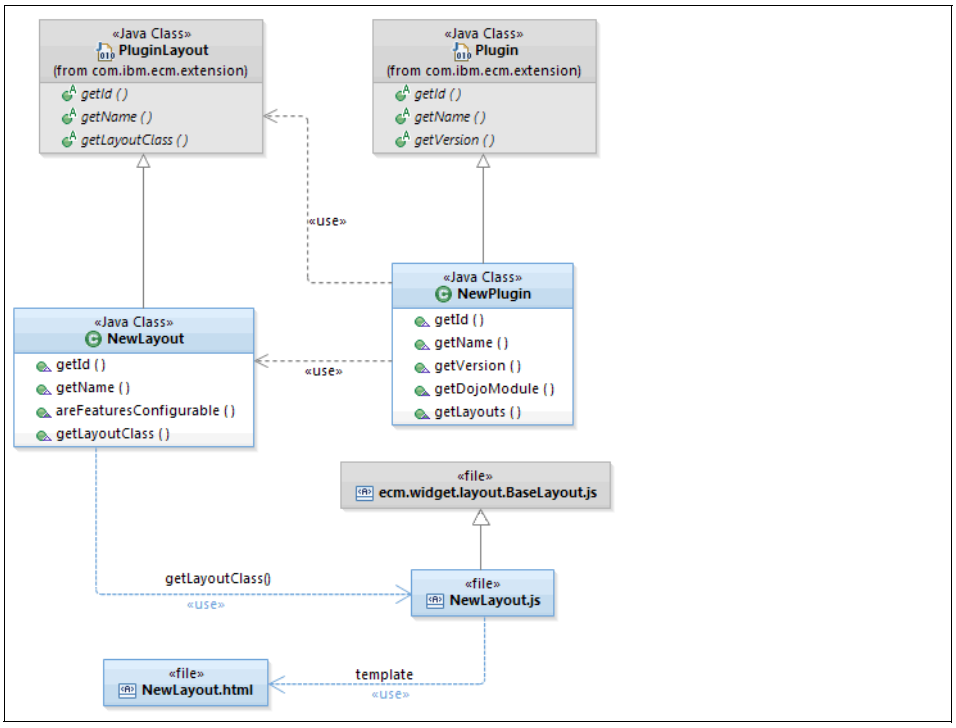



*Figure 1-14 Class diagram of a layout*

The core component for a new layout is the JavaScript file, which defines the user interface of the new layout. It must extend `ecm.widget.layout.BaseLayout` class or one of its subclasses, such as `ecm.widget.layout.MainLayout` or `ecm.widget.layout.NavigatorMainLayout`. The BaseLayout class provides the following functionality for your subclasses:

- Ability to enable demo and debug modes
- Processing and error dialog setup
- Login and logout handling
- Session timeout handling

The MainLayout class adds the visual part containing the banner, message bar, global toolbar and the LaunchBarContainer that contains the feature list on the left side and the LaunchBarContentArea for the content of the feature.

The NavigatorMainLayout class provides additional interactions specific to IBM Content Navigator and performs repository syncing between Browse and Search features.

Implementing a new layout can be summarized with the following steps:

1. Create a new widget that extends ecm.widget.layout.BaseLayout or one of its subclasses.
2. Create a Java class that extends the PluginLayout class, and implement the getLayoutClass() method to return the name of the widget class that was created in step 1.
3. Implement the getLayouts() and getDojoModule() methods in the Plugin Java class.

## Response filter and request filter

When executing a service call, a request is sent to the service and the response is sent back from the service to the client tier. These responses and requests are formatted in JSON. In general, the IBM Content Navigator framework creates the requests and responses in a standard JSON format that is defined by the service call and the framework.

In some instances, you might want to modify the requests and responses to modify the data that is being sent to or returned from the service. For example, you might want to check the documents that are included in the request and perform an extra level of security check based on external information. For the response, you might want to implement extra error handling code when the service is not able to perform a specific request. You can also define special formatting for dedicated columns of a result set through modifying the JSON output by using special Dojo classes. A request or response filter stands between the client and the service, and modifies the responses and requests as needed. A filter is always assigned to one service or a list of services within IBM Content Navigator.

Provided services are listed in the `struts-config.xml` file, which is located in the following directory:

`<WAS_InstallPath>/profiles/<nameOfProfile>/installedApps/<nameOfCell>/navigator.ear/navigator.war/WEB-INF`

In the <action-mappings> section of the `struts-config.xml` file, a list of actions is defined that can be filtered. The name of the action that must be returned by the getFilteredServices() method is the <action path> tag.

Figure 1-15 shows the required components to create request filter plug-in and response filter plug-in.

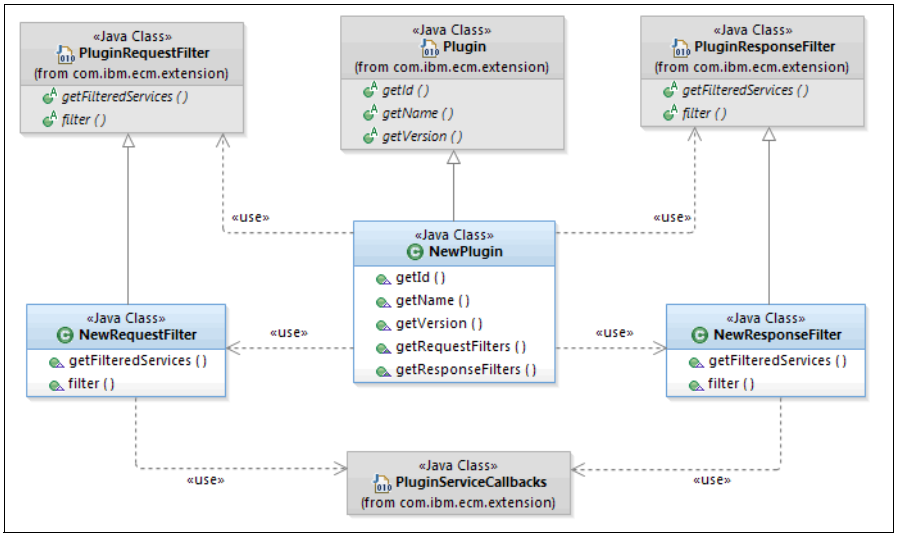



*Figure 1-15 Class diagram of RequestFilter and ResponseFilter*

A new request filter is created by extending the PluginRequestFilter class and implementing the following methods:

- The getFilteredServices() method, to define when the filter should be applied
- The filter() method, to define the behavior of the filter

A response filter is similar to the request filter definition. The difference is that, instead of extending the PluginRequestFilter class, the PluginResponseFilter class must be extended and instantiated within the Plugin class.

**Important:** A format of the JSON messages exchanged between client and server is subject to change from release to release. You might need to update your code of request and response filters if newer version of IBM Content Navigator is released.

Implementing a new request and response filter can be summarized with the following steps:

1. Define the request and response filter by extending the PluginRequestFilter or PluginResponseFilter class, and implement the getFilteredServices() and filter() methods.
2. Implement the getResponseFilter() or getRequestFilter() method in the Plugin Java class.

## 1.4.4 Content Navigator JavaScript API

Content Navigator JavaScript API are provided in the JavaScript Modeling Library and JavaScript Widget Library.

### JavaScript Modeling Library

The classes in the modeling library provide the business logic and data for IBM Content Navigator. These classes are used by the widgets to access and represent data in the content servers and in the IBM Content Navigator configuration. Figure 1-16 on page 48 shows the hierarchy of the primary classes in the modeling library.

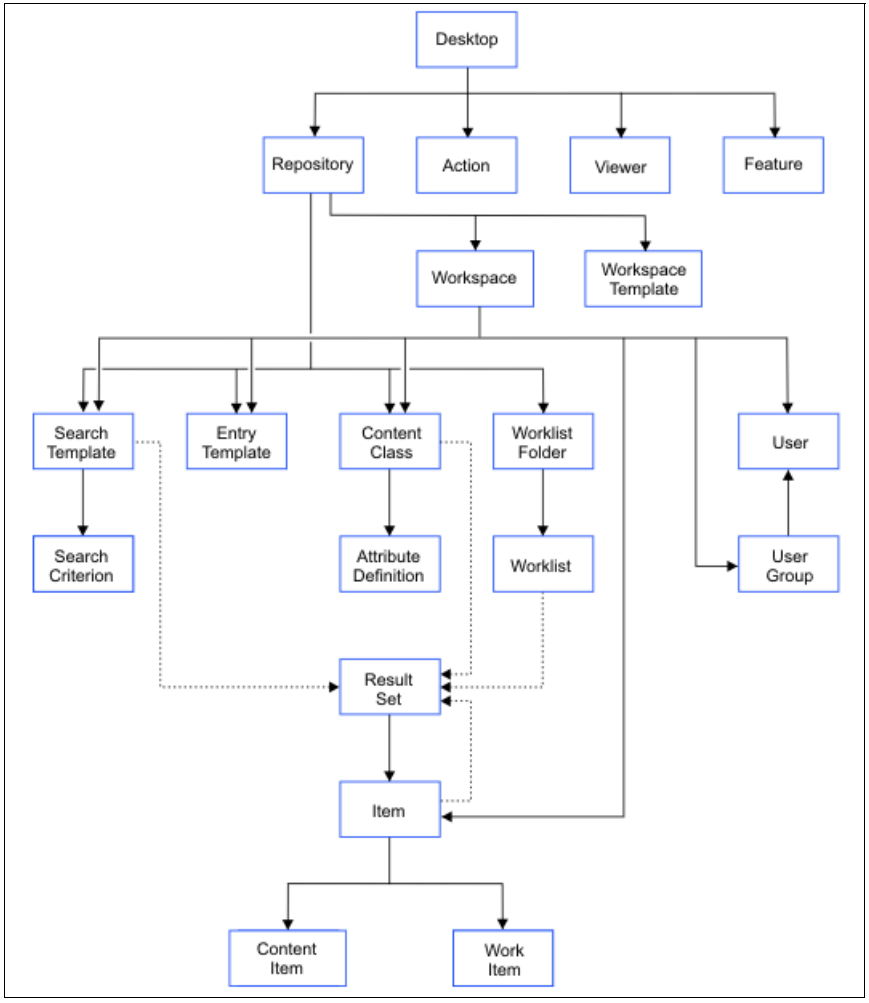



*Figure 1-16 JavaScript model diagram*

As shown in this diagram, an instance of the Desktop class encompasses the other objects in the model. The Desktop object specifies the repositories, features, actions, and viewers to which a set of users will have access.

An instance of the Repository class represents a specific repository. The repository can be an IBM Content Manager, IBM Content Manager OnDemand server, IBM FileNet Content Engine object store or also a CMIS repository.

With the Repository object, users can access and perform actions on repository objects. These objects are also represented by classes in the modeling library.

### *SearchTemplate class*

This class represents a search that is stored in the repository. In an IBM Content Manager or IBM FileNet P8 repository, a SearchTemplate object represents a saved search. In an IBM Content Manager OnDemand repository, a SearchTemplate object represents a folder. With a SearchTemplate object users can enter or modify the criteria that is used to perform a search. The criteria is represented in the model library by the SearchCriterion class.

### *EntryTemplate class*

This class represents an entry template that is stored in an IBM FileNet P8 repository. With an EntryTemplate object, users can create a document, folder, or custom object. An EntryTemplate object also provides the default values for the destination folder, properties, and security.

### *ContentClass class*

This class represents a document or folder class in a IBM FileNet P8 repository or an item type in an IBM Content Manager repository. With a ContentClass object, users can access the item and to edit the properties and attributes of the item. Each property or attribute is represented by an instance of the AttributeDefinition class that contains information about the property or attribute, such as type and allowed values.

### *WorklistFolder class*

This abstract class represents a collection of work lists, which are sometimes called in-baskets or inboxes. The subclasses of the WorklistFolder class represent collections of the types of work lists and in-baskets. For example, the ProcessApplicationSpace class represents a collection of process roles that determine who can access an IBM FileNet P8 process application. The ProcessRole class represents a role that is defined on an IBM FileNet P8 server. An instance of this class determines who has access to the in-baskets that are defined in an application space.

### *Worklist class*

This class represents a single work list. With a Worklist object, users can process the work items that are assigned to them. The model includes the ProcessInBasket subclass for IBM FileNet P8 repositories.

### *Teamspace class*

This class represents a teamspace. A Teamspace object provides users with the ability to organize and share the content that a team needs to complete its tasks. The individual users and groups who belong to a teamspace are represented in the model by the User and UserGroup classes.

### *TeamspaceTemplate class*

This class represents a teamspace template. With a TeamspaceTemplate object, users can set predefined options for creating a teamspace.

### *ResultSet class*

This class represents a set of items that are returned by a search or the content of a folder or worklist. With a ResultSet object, users can locate and select documents, folders, or work items.

An individual item in a ResultSet object is represented in the model by the Item class or one of its subclasses:

- The ContentItem class represents a document, folder, or other content item in the repository.
- The WorkItem class represents a workflow item.

### *Other classes*

Other classes in the modeling library support the classes that are shown in the diagram (Figure 1-16 on page 48). For example, the model includes the following classes:

- The SearchTemplateFolder class and TeamspaceFolder class represent collections of search templates and teamspaces. These abstract classes provide collections for items such as the recent folders or all folders that are displayed in navigation trees in the user interface.

  For an IBM Content Manager OnDemand repository, a SearchTemplateFolder object represents a cabinet.
- The Request class represents a request that is made to an IBM Content Navigator service.
- The PropertyFormatter class represents the formatting that is applied to properties when they are displayed. This class can be extended or overridden to provide custom formatting of certain types of properties.
- The _ModelStore class and the classes that are suffixed with TreeModel represent Dojo data stores and trees. These classes can be used with Dojo dijit classes to populate widgets with data.

## JavaScript Widget Library

The classes defined in the widget library provide the visual components for rendering the data obtained from JavaScript model. These classes extend the dijit/_Widget class defined in the Dojo 1.8.4 library and altogether build the user interface of the IBM Content Navigator.

The “root” widget that renders the IBM Content Navigator desktop is the DesktopPane widget. However, the layout of the widgets on the page is defined by NavigatorMainLayout class, which also provides access to all available features. Figure 1-17 on page 52 shows the hierarchy of primary components that defines standard features in the widget library.

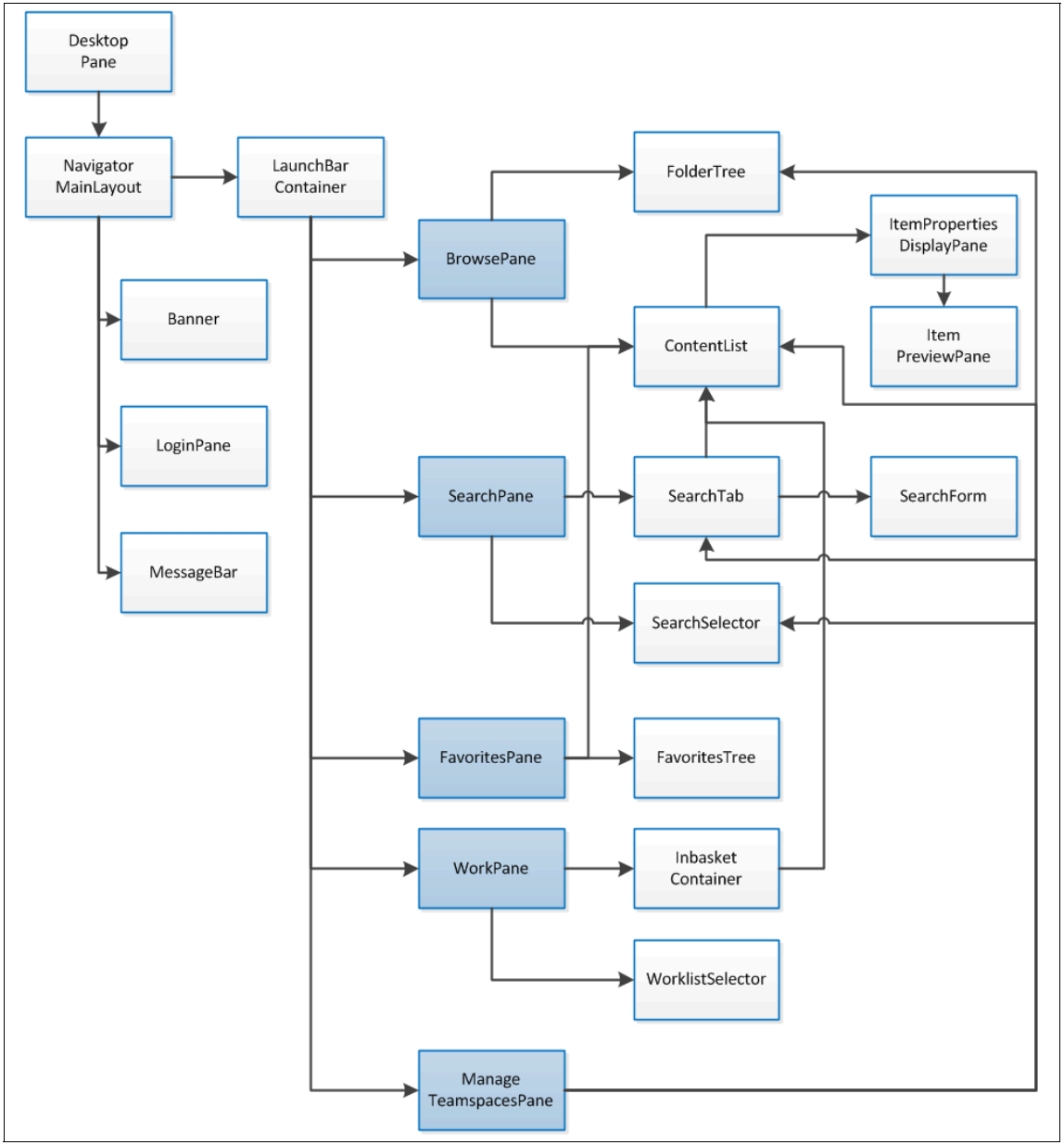



*Figure 1-17 Hierarchy diagram of the features in the JavaScript Widget Library*

The NavigatorMainLayout defines the top level layout of the IBM Content Navigator. In addition to banner, message box, and global toolbar, it also defines the main working area (LaunchBarContainer) where all available features are rendered. Figure 1-18 shows the layout of primary widgets defined in the NavigatorMainLayout class.

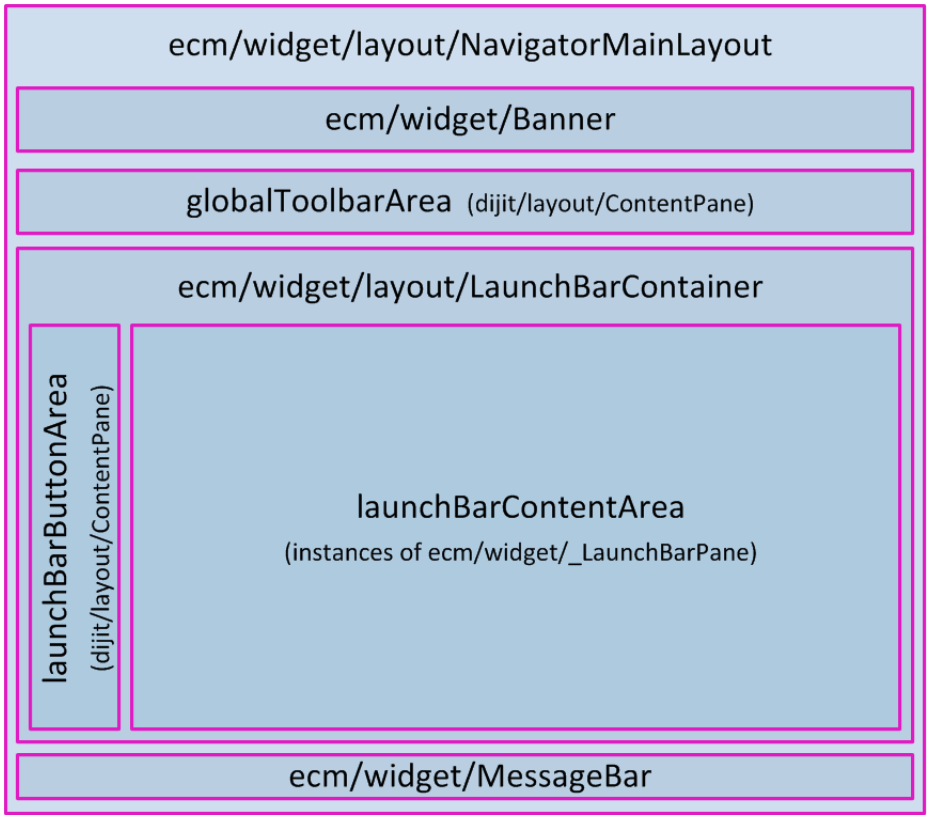



*Figure 1-18 Layout of the NavigatorMainLayout class*

For your convenience, we include an overview of relevant widgets (based on information center) in this section. The widgets are divided in groups by functionality:

- Widgets package (ecm.widget)

  The classes in the package define widgets that you use to create and edit repository objects such as documents, folders, and work items. This package also includes classes that define common user interface components such as buttons, menus, and tabs.

- Administration widgets package (ecm.widget.admin)

  The classes in the package define the widgets that make up the IBM Content Navigator administration tool. You can use these widgets to customize the administration user interface.

- Content list widgets package (ecm.widget.listView)

  The classes in the package define widgets, modules, and decorators that are used to construct the content list seen in IBM Content Navigator. The content list is used to provide lists of repository items, favorites, work items, and more.

- Dialog box widgets package (ecm.widget.dialog)

  The package contains classes that define the dialog boxes that are used in the IBM Content Navigator web client.

- Layout widgets package (ecm.widget.layout)

  The package contains classes that define components of the web client layout.

- Process widgets package (ecm.widget.process)

  The package contains classes that are used to build custom launch processors and step processors for an IBM FileNet P8 workflow.

- Search widgets package (ecm.widget.search)

  The classes in the package define widgets that you use to search repository objects such as documents, folders, and work items.

- Teamspace builder widgets package (ecm.widget.teamspaceBuilder)

  The package contains the widgets that create and edit teamspaces and teamspace templates. In IBM Content Navigator, these widgets are used for the teamspace builder.

- Viewer widgets package (ecm.widget.viewer)

  The package contains classes that are used to view the document content.

New custom visual widgets can be created in the visual layer. Creating widgets can be done either by inheriting or overwriting an existing widget from the IBM Content Navigator visual widget library, or by writing a completely new widget. Because a widget is only the visual representation of content (for example, list of documents and statistics), using the underlying modeling tier can be helpful to access the data and implement the business logic.

## 1.5 IBM Content Navigator samples

The IBM Content Navigator software package includes several samples that cover the major use cases for you custom development. You can use these samples as a base for creating custom applications.

### 1.5.1 Sample external data service

The sample external data service (EDS) implements an EDS REST interface as Java Platform, Enterprise Edition application deployable on application server. It uses the file system as an external source and allows you to create a file for your class or workflow step, where you can specify the “properties” part of the JSON response returned from your EDS interface.

The sample EDS also extends the EDS JSON format with these capabilities:

- Definition of dependent choice list values for a property by specifying dependentOn and dependentValue attributes
- The validateAs attribute that allows you to specify two types of custom validations:
  - NoThrees: Validation for occurrence of the “3” character in a user-entered value.
  - Required: Check for empty value in a user-entered value.
- The timestamp attribute specifying that the current time will be saved as attribute value, without notifying the user.

The sample web pages are available in the following directory:

*ECMClient_installdir*\samples\sampleEDSService

### 1.5.2 Sample web pages

The sample web application demonstrates the use of the IBM Content Navigator Toolkit. The pages demonstrate various aspects of the IBM Content Navigator toolkit. The samples are divided into three parts, each explaining a separate approach for developing a custom user interface:

- How to work with JavaScript model classes.
- How to use stand-alone widgets in your application.
- How to build complete new application using the toolkit.

The sample web pages are available in the following directory:

*ECMClient_installdir*\samples\ samplePages

Deploy the included WAR file into the application server with IBM Content Navigator in order to get it working.

### 1.5.3 Sample plug-in application

The sample plug-in implements a plug-in that demonstrates several extension capabilities available to plug-in writers. The files in this sample application show how to implement various extension points in IBM Content Navigator plug-in development. These samples demonstrate the following tasks:

- How to use action handling for custom file uploads.

  This functionality in the sample includes single file upload control from the client, server access of the uploaded document in the plug-in action, and sample service for file manipulation. Look for the following file:

  `SamplePluginFileUploadAction.java`

- How to change property formatter and property editor.

  It includes sample formatter that will render createdBy and modifiedBy attributes as email links in ItemPropertiesDisplayPane. It also includes sample property editor that adds a button next to DocumentTitle text field and makes the DocumentTitle attribute read only. Look for the following file:

  `SamplePluginOpenClassResponseFilter.java`

- How to change property decorator in content list widget.

  Look for the `SamplePluginResponseFilter.java` file.

- Two alternatives for how to create a custom viewer:
  - By using URL pattern. Look for the `SamplePluginViewerDef.java` file.
  - By implementing custom viewer widget. Look for `SamplePluginImageViewerDef.java`

- How to extend the item properties pane, used in the property editor, to add a custom section. Look for `SampleItemPropertiesPaneExtension.js` file.

- How to create feature and service for custom search.

  It includes widget for entering custom search query and services for IBM FileNet P8 and IBM Content Manager repositories. Look for the `SamplePluginFeature.java` file.

The files for the sample plug-in are available in the following directory:

*`ECMClient_installdir`*`\samples\samplePlugin`

The `SamplePlugin.jar` file that you can deploy by using the IBM Content Navigator administration tool is available in the following directory:

*ECMClient_installdir*\navigator

### 1.5.4 Sample mobile application

This sample demonstrates how to create a stand-alone mobile application that requires content management capabilities. These samples were developed using the IBM Worklight Studio, and the sample mobile application project is provided in that format. The principles and code provided can be adapted to other development environments, and do not require any server-side prerequisites, except for a properly configured IBM Content Navigator server. The sample mobile application covers following functionality:

- Browse a repository.
- Search.
- Work with favorites.
- Use IBM Datacap Taskmaster Capture.
- Preview documents.
- Open documents in other applications.
- Check out documents.
- Check in documents from other applications by using “Open in” function.
- Edit metadata of documents.
- Persist application data.

The sample Worklight project is packed in the `SampleMobileApp.zip` file, located in the following directory:

*ECMClient_installdir*\samples\

### 1.5.5 Sample mobile plug-in

This sample, which provides a mobile web layout for IBM Content Navigator, enables users to access the server by using a web browser on a mobile device. This sample demonstrates how mobile access to the IBM Content Navigator server can be supported without installing a mobile app. The `sampleMobilePlugin.jar` file is a standard IBM Content Navigator plug-in that is deployed to IBM Content Navigator and applied to a desktop.

The sample layout contains the following functionality:

- Browse a repository
- Search
- Work with favorites
- Preview documents

- Open documents in other applications (Open in)
- Check out documents
- Edit metadata of documents
- Persist application data

The sample plug-in JAR file SampleMobilePlugin.jar, is available in the following directory:

*ECMClient_installdir*\samples

For a list of samples provided with IBM Content Navigator, see 1.5, “IBM Content Navigator samples” on page 55. See 1.6, “Samples we developed for this book” on page 58 for a list of samples.

## 1.6 Samples we developed for this book

Several examples are described in this book and use various development options and extension points that IBM Content Navigator provides. Each chapter describes the scenario when and how the sample code can be used. The following list summarizes the extension points used in the samples:

- Chapter 4, “Developing a plug-in with basic extension points” on page 117:
  - Plugin/Action: Opens a new “create folder” dialog that extends existing AddContentItemDialog and calls a custom service after creating the folder.
  - Plugin/Service: Creates subfolders for a specified folder based on template configuration.
  - Plugin/Action: Opens a feature and preselects a folder.
  - Plugin/Feature: Extends standard browse feature.
- Chapter 5, “Building a custom repository search service” on page 173:
  - Plugin/Feature: Displays searchbox and standard ContentList widget, allowing custom repository search with SQL like queries.
  - Plugin/Service: Performs SQL like query in the repository.
  - Plugin/RequestFilter: Modifies the ‘/p8/continueQuery’ request to provide paging functionality for the custom search ContentList widget.
- Chapter 6, “Creating a feature with search services and widgets” on page 211:
  - Plugin/Feature: Displays virtual folder structure using the Tree digit and ContentList widget.

- Chapter 7, “Implementing request and response filters and external data services” on page 233:
  - EDS: Reads property configuration including formatting, validation and choicelist values from database.
  - Plugin/ResponseFilter: Modifies the ‘*/openContentClass’ request to reorder the properties of specified class displayed in IBM Content Navigator.
  - Plugin/ResponseFilter: Modifies the cardinality of specified property in ‘*/openContentClass’ request to provide custom property editor with search dialog.
  - Plugin/Widget: Custom property editor that allows searching in large choice lists.
- Chapter 8, “Creating a custom step processor” on page 277:
  - Step processor: Embeds the document viewer into the user interface of the workflow step.
  - Plugin/Action: Opens the document in “embedded” viewer if it is available on the page.
- Chapter 10, “Customizing built-in viewers and integrating third-party viewers” on page 365:
  - Plugin/ViewerDef: Uses plug-in service to open a viewer.
  - Plugin/Service: Stores and loads plug-in configuration and opens the Snowbound VirtualViewer by redirecting the browser.
- Chapter 9, “Using Content Navigator widgets in other applications” on page 297:
  - Plugin/Layout: Displays just the ContentList widget.
  - Integration: Shows how to integrate IBM Content Navigator into stand-alone application, into Microsoft SharePoint and into IBM WebSphere Portal.
- Chapter 11, “Extending solutions to mobile platform” on page 413:
  - Mobile development: Shows how to add Work list feature into the mobile application.
- Chapter 12, “Extending Profile Plug-in for Microsoft Lync Server” on page 437:
  - Plugin/ResponseFilter: Modifies ‘*/search’ and ‘*/openFolder’ requests to provide custom decorator for certain columns.
  - Plugin/Services: Acts as a proxy for the requests to the Microsoft Lync Server.

## 1.7 Conclusion

This chapter describes the architecture of the IBM Content Navigator and available programing interfaces. It also gives an overview of all the available configuration and development options that IBM Content Navigator provides.

The following chapters describe these options in detail with sample codes as summarized in 1.6, “Samples we developed for this book” on page 58.

2

# Customizing desktop appearance

The IBM Content Navigator Administration Desktop provides an easy way to customize the user interface. This chapter describes how the IBM Content Navigator user interface can be configured by changing the desktop appearance. It shows how to change logos, colors, and icons to make the desktop match your corporate standards.

This chapter covers the following topics:

- Customizing the desktop appearance
- Adding a logo and banner color
- Adding login notes
- Adding password rules

© Copyright IBM Corp. 2012, 2014. All rights reserved.

## 2.1 Customizing the desktop appearance

The visual characteristics of IBM Content Navigator are displayed using a customized theme. The theme determines the overall appearance, such as fonts and colors. Although altering the theme is possible, it is currently not a supported option for IBM Content Navigator. You can make changes to the desktop without altering the theme. For example you can make the following changes:

- Add your company logo to both the login page and banner of the desktop.
- Add useful notes to the login page.
- Alter the color of the desktop banner.
- Add or remove features from the desktop.
- Alter the menu options on both toolbar and context menus.
- Change icons.
- Change the lists of properties available for search and browse.
- Alter the labels used in the desktop.

The following sections describe how to change the visual appearance of the desktop by adding a logo, changing the banner, and adding login notes and password rules.

## 2.2 Adding a logo and banner color

New logos must be referenced by a URL. Create a new web application and add your logos to the Web Content folder. If you already created a separate web application for plug-ins, reuse it for external content. Deploy the application, ensuring that it is available to all servers where IBM Content Navigator is installed.

The IBM banner logo used by IBM Content Navigator is 43 x 16 (width x height) pixels. Depending on the background color and other factors, the banner width can accommodate a larger image. The logo should not be larger than 200 x 150 pixels and is recommended to use transparent background.

The login logo can be larger than the banner logo. The example login logo used later in this section is 150 x 112 pixels.

When you use a URL to reference the external content, such as a logo or an HTML page, you can use a relative path if the content is in a web application that is located on the same web server as IBM Content Navigator. The examples in this chapter refer to `/customConfig`. If you have deployed it in a highly available configuration, use a fully qualified path that will be available to all instances of IBM Content Navigator.

For example, use the following URL:

`http://<server_name>/customConfig`

For our example, `ecmclient` is the DNS resolvable name:

`http://ecmclient/customConfig`

Use the Administration Desktop or administration feature from your desktop to edit a previously saved desktop. For our example, we edit the desktop named Data Management.

To configure the desktop to access a new log and external content, complete the following steps:

1. Select **Desktops**.
2. Select your desktop from the list of available desktops, in our case **Data Management**
3. Click **Edit** and select the **Appearance** tab.

   To use your company logo on the desktop banner, enter a URL to access the logo.

Figure 2-1 shows our example Data Management desktop setup.

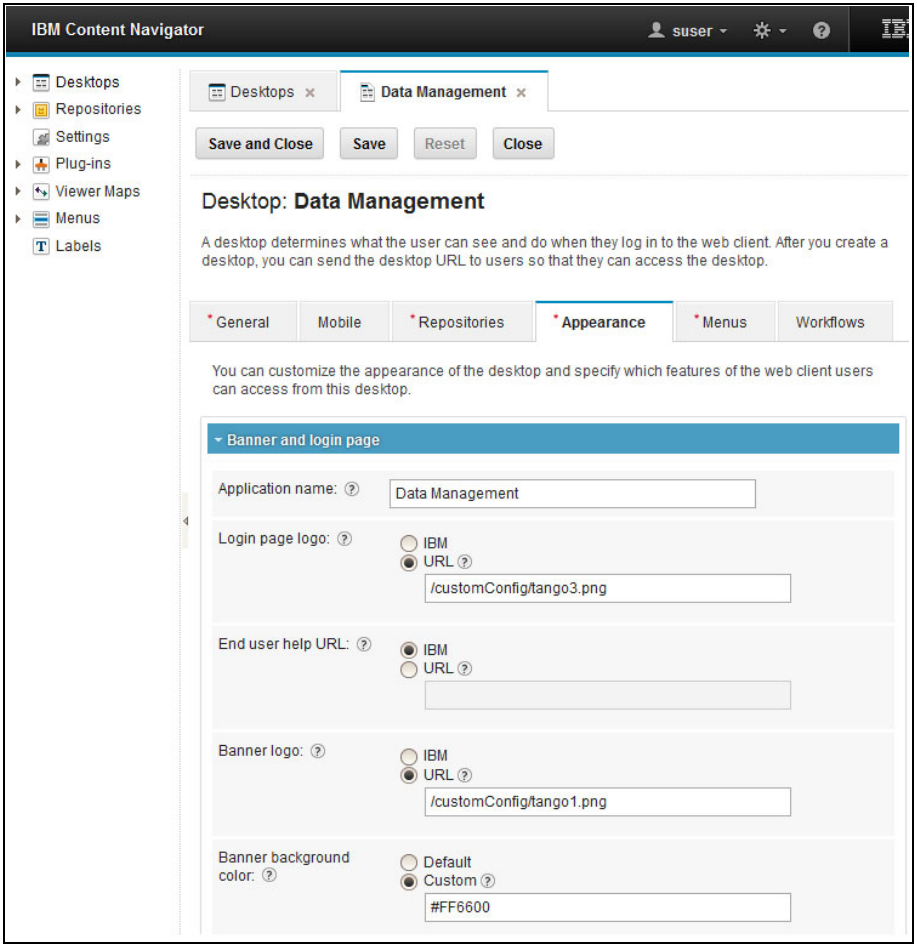



*Figure 2-1 Configure the desktop appearance*

Enter the following information:

– URL for the banner logo.

  For our example, we enter: `/customConfig/tango1.png`

– URL for the login page logo.

  For our example, we enter: `/customConfig/tango3.png`

– Color code for the banner background color.

  For our example, we enter: `#FF6600`

4. Save the desktop.

Figure 2-2 shows the result of our new desktop appearance.

**Color code**: Colors must be expressed as either three-character or six-character hexidecimal codes.

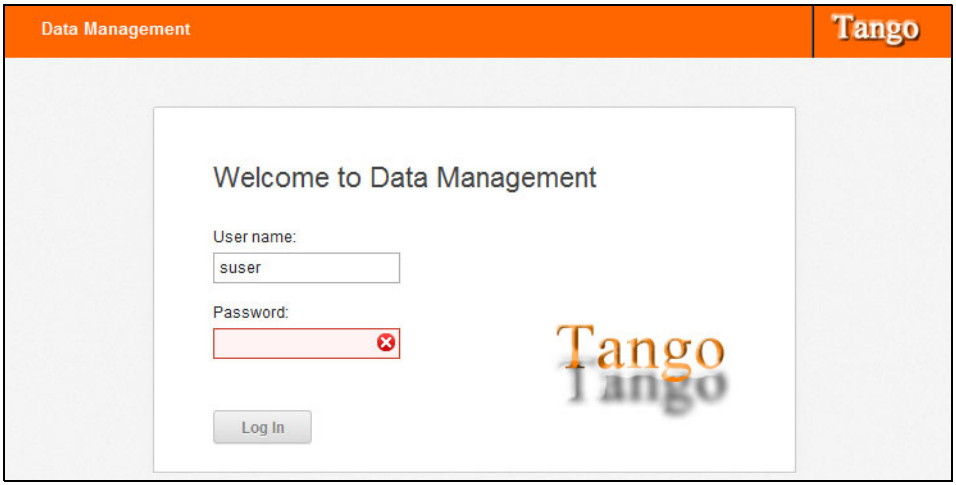



*Figure 2-2 Testing the desktop appearance*

## 2.3 Adding login notes

Adding content to the login page provides a method for communication of the latest information or news to users when they access IBM Content Navigator. This information can include guidance about the use of the system, the hours of operation, and help desk contact procedures. In addition, you can notify users of planned downtime for system maintenance on the login page.

To add login page content, prepare a new HTML file that contains the information that you want to display to users. Add the file to a web application that is available to all systems where IBM Content Navigator is installed.

To add content to the login page, use the Administration Desktop and follow these steps:

1. Select **Desktops**.
2. Select the desktop to be modified.

   For our example, we select **Data Management**.
3. Click **Edit** and select the **Appearance** tab.

4. Enter a URL for the HTML page that contains the login content.

   For our example, we enter: `/custom/Config/loginNotes.html`

   Figure 2-3 shows the login page setup.

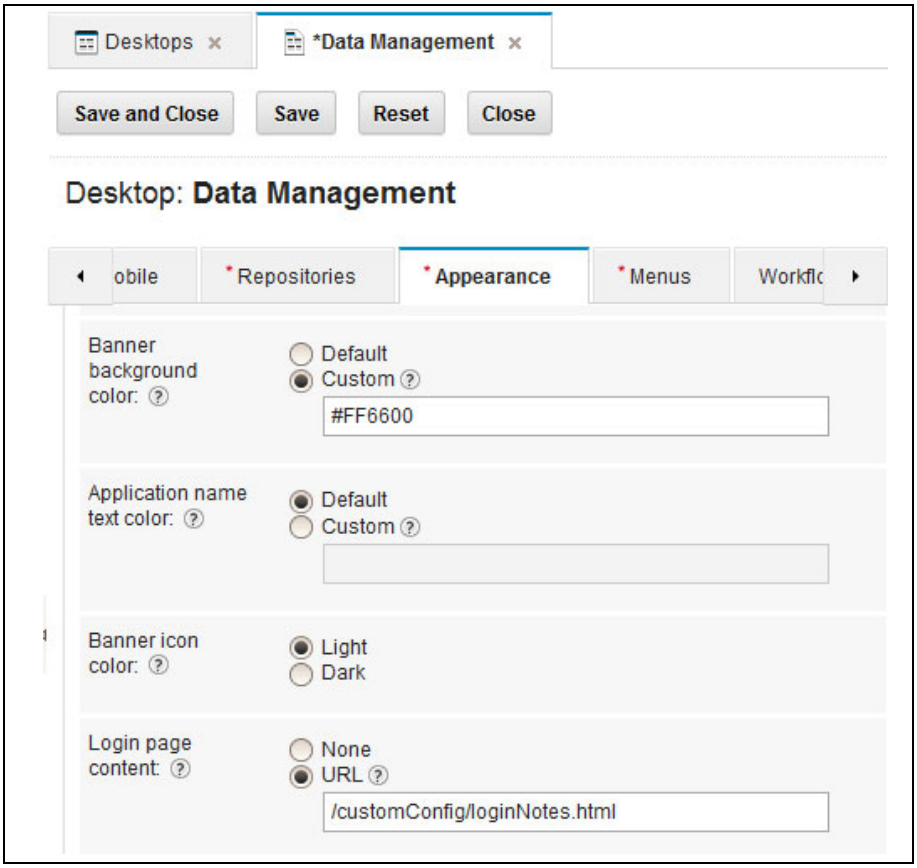



*Figure 2-3 Configure login page content*

**Note:** If you use an image in the login content, make sure that the `src` attribute of the <img> tag uses a fully qualified path to location of the image file, as the following example shows:

```
<img border="0"
src="http://ecmclient:9080/customConfig/tango3.png" width="150"
height="112">.
```

5. Save the configuration.

Figure 2-4 shows our example of the revised login page with the additional login notes.

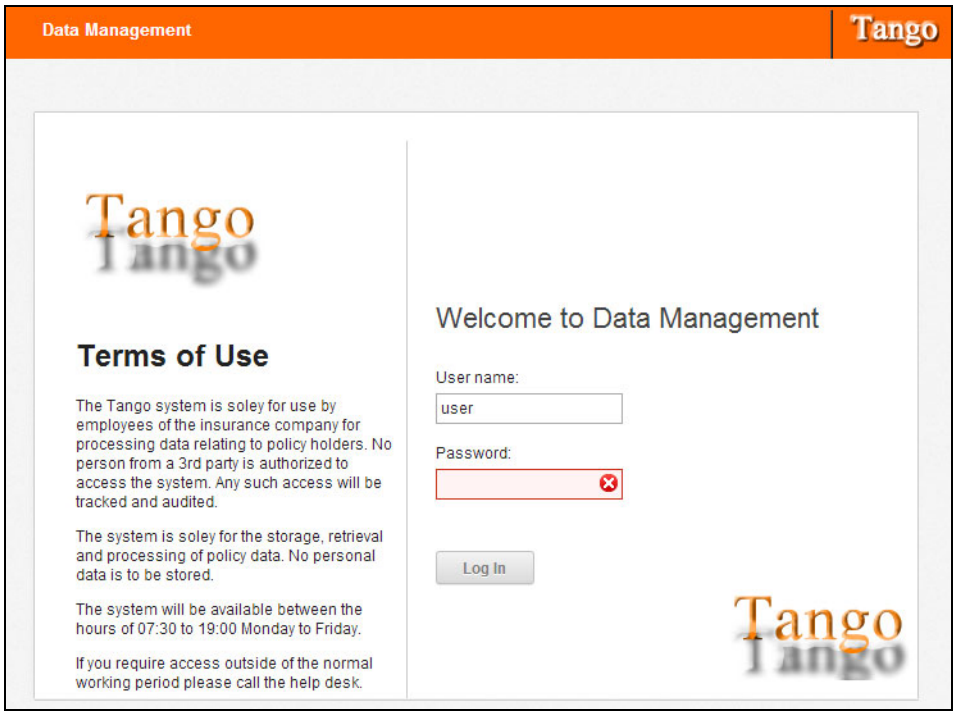



*Figure 2-4 Displaying additional content on the login page*

For additional information about changing the appearance of IBM Content Navigator, see the IBM Knowledge Center for IBM Content Navigator.

Example 2-1 shows the `loginNotes.html` file that we modified.

*Example 2-1 The loginNotes.html file we modified*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>loginNotes</title>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
</head>
<body>
<p><div> <img border="0"
src="http://localhost:9080/customConfig/tango3.png" width="150"
height="112"><span><h1>Terms of Use</h1></span></div>
The Tango system is soley for use by employees of the insurance company
for processing data relating to policy holders. No person from a 3rd
party is authorized to access the system. Any such access will be
tracked and audited.
<p>The system is soley for the storage, retrieval and processing of
policy data. No personal data is to be stored.</p>
<p>The system will be available between the hours of 07:30 to 19:00
Monday to Friday.</p>
<p>If you require access outside of the normal working period please
call the help desk.</p>
</body>
</html>
```

## 2.4 Adding password rules

IBM Content Navigator enables users to change their passwords. Each company has its own password rules that are in force at the time. To communicate the rules information to users, you can add a password rules information page to the user interface so users can click the information link and read the rules for passwords.

**Note:** Systems that are configured to use LDAP for authentication, such as IBM FileNet Content Manager, are not allowed to change passwords by using IBM Content Navigator.

To include the password rules information from the desktop, you first create an HTML file that contains the information describing your password rules. Then, you add the file to your web application. The example in this section uses the `passwordRules.html` file.

From the Administration Desktop, add the password rules information to a desktop as follows:

1. Select **Desktops**.
2. Select the appropriate desktop from the list.

   For our example, we select **Data management**.
3. Click **Edit** and select the **Appearance** tab.
4. Enter the URL for your external content in the Password rules field.

   For our example, we enter: `/custom/Config/passwordRules.html`. See Figure 2-5.

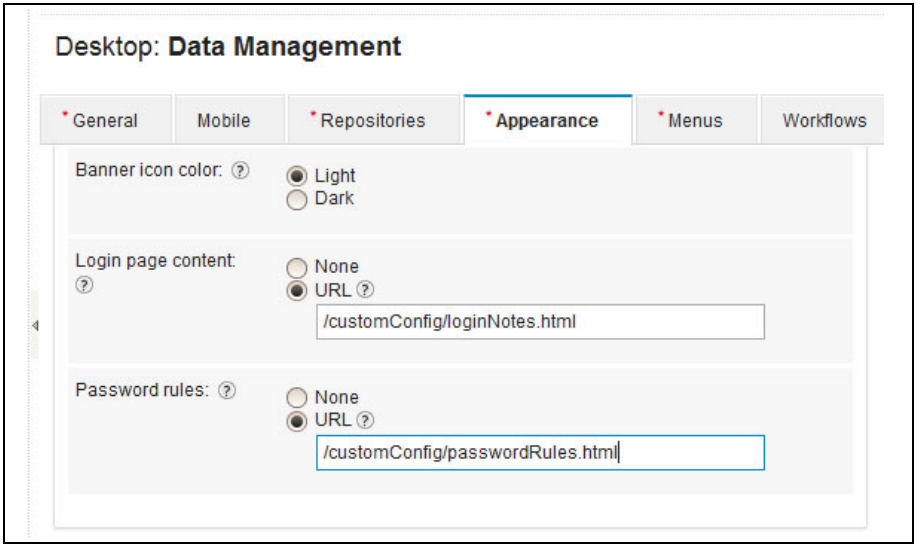



*Figure 2-5 Configure desktop password rules*

5. Save the configuration.

Example 2-2 shows the content of the `passwordRules.html` file.

*Example 2-2 Content of the passwordRules.html file*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>passwordRules</title>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
</head>
<body>
<p>
<div> <img border="0"
src="http://ecmclient:9080/customConfig/tango3.png" width="150"
height="112"><span><h1>Password rules</h1></span></div>
Password for the Tango system must conform to the following rules:
<p>Minimum length 8 characters</p>
<p>Minimum of 1 digit and 1 uppercase alpha character to be used</p>
<p>Password to be changed every 90 days, no password re-use.</p>
<p>If you require assistance please call the help desk.</p>
</body>
</html>
```

## 2.4.1 Testing the password rules

To test your password rules, complete the following steps:

1. Launch IBM Content Navigator by using the URL with you desktop name:

   `http://<server_name>:<port>/navigator/?desktop=DataManagement`

   For our example, we enter this URL:

   `http://ecmclient:9080/navigator/?desktop=DataManagement`

2. Use the user command tool, which you access by clicking the down arrow next to you user name, and select **Change Password**. See Figure 2-6 on page 71.

*Figure 2-6 Using the User Command Tool*

The Change Password dialog opens (Figure 2-7).

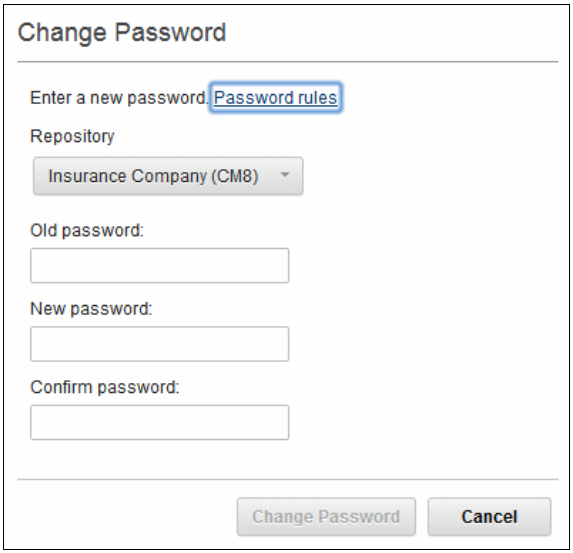



*Figure 2-7 Change the password dialog*

3. Select the **Password rules** link to display your page.

Our page looks like the example in Figure 2-8.

*Figure 2-8 A sample password rules display*

For additional information about changing the appearance of IBM Content Navigator, see IBM Knowledge Center for Content Navigator.

## 2.5 Conclusion

This chapter shows how you can change desktop appearance by changing desktop configuration using administration tools without coding.

If you want to make other types of modification or customization, review Chapter 1, “Extension points and customization options” on page 3 for the available customization and extension options. Then, follow Chapter 3, “Setting up the development environment” on page 73 to set up your environment, and other chapters in the book for examples of making those changes.

3

# Setting up the development environment

This chapter describes how to set up the development environment for IBM Content Navigator to customize and extend the IBM Content Navigator. It shows how to create a new plug-in project with an Eclipse wizard and also includes references for testing and debugging your application. At the end of this chapter, you should have an integrated development environment (IDE) that is ready to use and a basic understanding of how to create and deploy a IBM Content Navigator plug-in.

This chapter covers the following topics:

- Prerequisites for plug-in development
- Setting up development environment
- Plug-in development
- Creating a new empty EDS project using the wizard
- Getting started with the SamplePlugin
- Building a plug-in JAR manually
- Debugging plug-ins

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# 3.1 Prerequisites for plug-in development

Extending and customizing IBM Content Navigator through plug-ins can be done with any Eclipse based development environment. This chapter outlines the recommendations regarding the development environment and additional extensions for the following development environments:

- IBM Rational® Application Developer Version 9
- Eclipse Software Development Kit

You can choose between both development environments. Because Rational Application Developer is based on Eclipse, the necessary features are included in both development tools. Rational Application Developer offers more features for Dojo and web development.

**Note:** This chapter does not describe how to install the development environment. It shows only which additional package can be useful for the development of extensions for IBM Content Navigator. Basic development experience with Eclipse or Rational Application Developer can be helpful.

## 3.1.1 IBM Rational Application Developer

IBM Rational Application Developer is a powerful and flexible platform for developing Java Platform, Enterprise Edition and plug-in projects. It provides wizards and plug-ins that simplify Java, Java Platform, Enterprise Edition and web development. Within Rational Application Developer, you *can* set up *or* integrate multiple application servers and deploy your workspace projects directly to these servers. This is a convenient feature for testing and debugging web applications in multiple application servers. In addition, Rational Application Developer offers powerful tools for developing web applications using the Dojo Toolkit. With it, you can construct HTML pages and Dojo widget HTML templates by dragging existing widgets into a page editor. A Rational Application Developer project can be set up to access the Dojo Toolkit and visual editor with a preview for what-you-see-is-what-you-get (WYSIWYG).

The online documentation at the following link has useful information about installing and developing with Rational Application Developer Version 9:

`http://pic.dhe.ibm.com/infocenter/radhelp/v9/index.jsp`

Use the latest Rational Application Developer Version. If it is not possible, use Rational Application Developer Version 8.5, at the minimum.

During the installation process of Rational Application Developer, you can choose various packages to be installed. Select the following packages during installation:

- Java EE and web service development tools
- Depending on your target WebSphere Application Server version:
  - WebSphere Application Server Version X.X development tools
  - Tools for developing of applications without a local Server installation
- Ajax, Dojo Toolkit, and HTML development tools
- JSP and servlet development tools
- JSF development tools
- XML development tools
- IBM Worklight Package (when you want to create mobile applications:)

**Note:** IBM Worklight Developer Edition for mobile development can be selected from your Rational Application Developer download page and can also be downloaded from the web and installed as Eclipse plug-in. For details regarding the installation and usage of Worklight, go to the following address:

`http://pic.dhe.ibm.com/infocenter/wrklight/v6r0m0/index.jsp`

## 3.1.2 Eclipse development environment

The Eclipse Software Development Kit (Eclipse SDK) is an open source software development platform. The Eclipse organization is created by a consortium of software development tool providers comprised of IBM and other leading tool providers. Eclipse SDK is free to use and has different packages depending on the type of development you want to do.

For developing code shown in this book, use the Eclipse IDE for Java EE Developers package. It includes the basic Java Platform, Enterprise Edition components, JavaScript, HTML, and CSS editors.

The installation package is at the following location:

`http://www.eclipse.org/downloads/moreinfo/jee.php`

We used the latest Eclipse Kepler Package based on Eclipse Version 4.3.1. If you want to use the IBM Worklight Eclipse plug-in for mobile development, you must use the Indigo or Helios Eclipse Package.

If you decide to use the Eclipse IDE for Java EE Developers, you can easily create new plug-ins for Content Navigator by creating new Java projects and

build up your external data service application with the Java Platform, Enterprise Edition components.

For development of Dojo components, you can use tools such as Maqetta:

http://maqetta.org/

For details about installing, using, and downloading Worklight, see the Worklight Information Center.

### 3.1.3 WebLogic and Eclipse

In addition to WebSphere Application Server, IBM Content Navigator also supports WebLogic Server as an application server. The following Oracle web page lists an installer that delivers an Eclipse environment and the WebLogic server environment. This enables developers to build web applications, and deploy and test them directly on the application server within the development environment.

http://www.oracle.com/technetwork/middleware/weblogic/downloads/wls-main-097127.html

At this web page, see the “Installers with Oracle WebLogic Server, Oracle Coherence and Oracle Enterprise Pack for Eclipse” topic.

The Eclipse package we use is based on the Kepler Eclipse release and contains special features to develop and deploy on WebLogic servers. This package also delivers the necessary Java Platform, Enterprise Edition components.

## 3.2 Setting up development environment

In this section, we describe the steps which are necessary to prepare your development environment to easily create new IBM Content Navigator plug-in and external data services (EDS) projects.

There is an Eclipse plug-in for IBM Content Navigator that can be integrated in your development environment. The Eclipse plug-in eases the creation of new Content Navigator plug-in projects and external data service projects.

The Eclipse plug-in can be downloaded from the additional material link that is associated with this book. Two Java Archive (JAR) files are included with the plug-in:

- `com.ibm.ecm.plugin.202.jar`
- `com.ibm.ecm.icn.facet.EDSPlugin.202.jar`

These two JAR files contain the extensions for Eclipse, enabling both a new project for IBM Content Navigator plug-in development and a new Eclipse web project facet for building EDS web projects.

### 3.2.1 Installing the Eclipse plug-in in base Eclipse environments

Follow these steps to install the Eclipse plug-ins:

1. Download the JAR files from the additional materials for this book (see Appendix D, “Additional material” on page 535).
2. Copy the JAR files in the Eclipse `dropins` directory.

   The dropins directory is under the Eclipse installation path, as in this example:

   ```
   C:/eclipse/dropins
   ```

   If you are running Eclipse for WebLogic, the installation directory might be in a directory similar to the following one:

   ```
   C:/Oracle/Middleware/Oracle_Home/oepe/eclipse/plugins
   ```
3. Restart Eclipse.

   Sometimes it is necessary to call Eclipse with the -clean parameter from the command line to make the changes active, as in the following example:

   ```
   C:/eclipse/eclipse.exe -clean
   ```

### 3.2.2 Installing the Eclipse plug-in in Rational Application Developer

Follow these steps to install the Eclipse plug-ins in Rational Application Developer:

1. Download the JAR files from the additional materials for this book (see Appendix D, “Additional material” on page 535).
2. Copy the JAR files in the plug-ins directory of Rational Application developer, as in this example:

   ```
   C:/Program Files/IBM/SDP/plugins
   ```
3. Restart Rational Application Developer.

### 3.2.3 Troubleshooting Eclipse plug-in installation

If you restart Eclipse and do not see the option to create an IBM Content Navigator plug-in project, then you probably do not have all the dependencies installed. Check the Eclipse logs for errors. You can find the Eclipse log under the menu **Window** → **Show View** → **Error Log.** The following error in the log is typical:

```
Unable to satisfy dependency from com.ibm.ecm.icn.plugin.202 to bundle
org.eclipse.wst.jsdt.core [1.1.202,2.0.0).
```

To resolve this, go to the Eclipse marketplace and download the missing dependency.

**Note:** If you have installed Rational Application Developer 8.5 or later, or the latest Java Platform, Enterprise Edition version of Eclipse, then all IBM Content Navigator Eclipse plug-in dependencies should already be installed.

### 3.2.4 Verifying Eclipse plug-in installation

In 3.2.1, “Installing the Eclipse plug-in in base Eclipse environments” on page 77 and 3.2.2, “Installing the Eclipse plug-in in Rational Application Developer” on page 77, we describe how to install the Eclipse plug-ins for Content Navigator plug-in development and the installation of the external data services (EDS) Eclipse extension in your development environment. In this section, we verify that the installation was successful by validating that the new menus are visible in your development environment.

To verify the installation of the Eclipse plug-in for Content Navigator plug-in projects, follow these steps:

1. Open your development environment, for example, Eclipse or Rational Application Developer.
2. Go to **File** → **New Project** and ensure that you can see the IBM Content Navigator folder. In this folder, you should see Content Navigator Plug-in, as shown in Figure 3-1 on page 79. If the entry is listed in the dialog, the Eclipse plug-in for Content Navigator plug-in installation was successful.

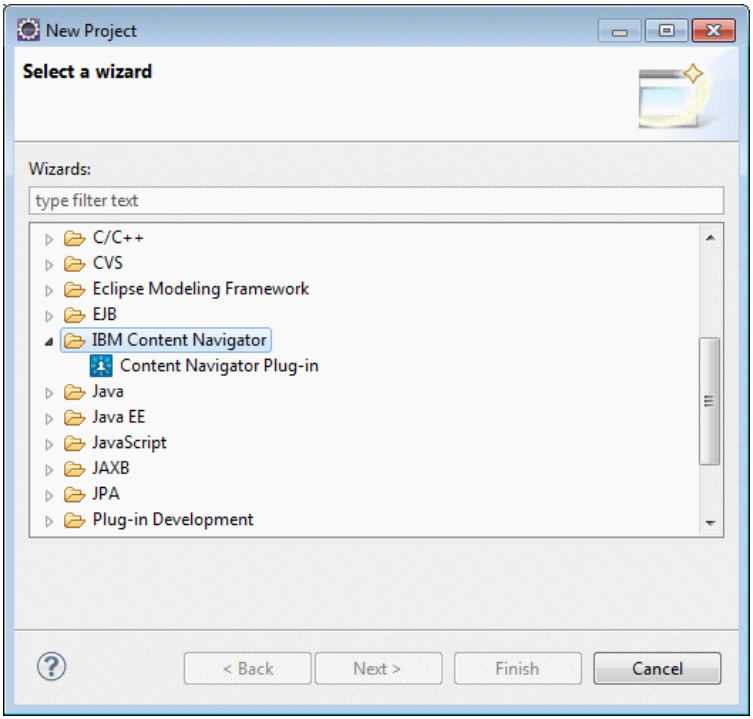



*Figure 3-1 New Content Navigator plug-in dialog*

3. Click **Cancel** to close the dialog. Being able to open the dialog is enough to verify that the installation was successful.

To ensure that the EDS Eclipse plug-in has also been installed successfully, follow these steps:

1. Open your development environment, for example Eclipse, if it is not already opened.
2. Go to **File** → **New Project** → **Dynamic Web Project**. In the wizard, ensure that you can select **IBM External Data Service Web Project** in the **Configuration** section, as shown in Figure 3-2 on page 80. If the entry is listed in the dialog, the plug-in for external data services was installed successfully.

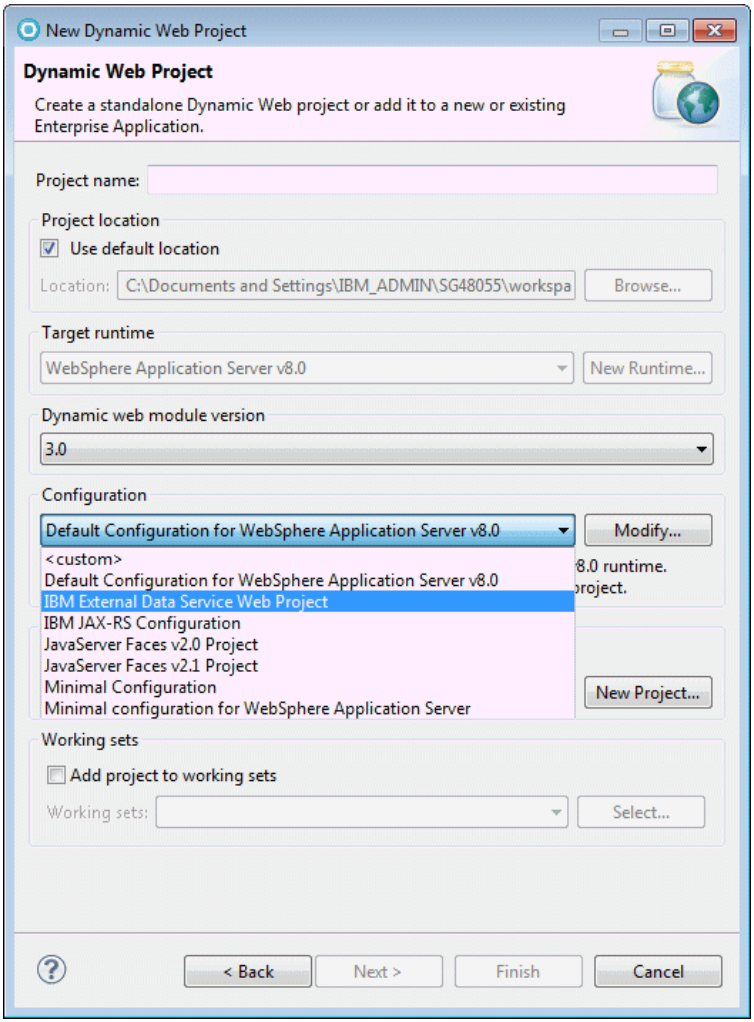



*Figure 3-2 New Dynamic Web Project for External Data Service*

We show how to install the Eclipse plug-in for Content Navigator plug-in and EDS web applications. At this point, you are ready to create your first Content Navigator plug-in project and external data service web project.

**Before you begin:** Read through Chapter 1, “Extension points and customization options” on page 3 to review the skillset that is required for plug-in development. If you are new to this type of development, go through the helpful links provided in the various sections of that chapter to gain necessary knowledge before you start setting up the environment for Content Navigator plug-in development.

## 3.3 Plug-in development

This section provides detailed information about creating, packaging, and deploying a Content Navigator plug-in project. We will describe the structure of a plug-in project and also provide information on how to deploy and test a Content Navigator plug-in.

A prerequisite for this section is that the Content Navigator Eclipse plug-in installed successfully as described in 3.2, “Setting up development environment” on page 76.

### 3.3.1 Creating a simple plug-in project using the wizard

This section describes how to create a new project in your development environment for customizing or extending IBM Content Navigator.

Before creating a new Content Navigator project, ensure that you have access to the `navigatorAPI.jar` file. The file is located in the `lib` folder of the Content Navigator installation directory, as in the following example:

```
C:\Program Files (x86)\IBM\ECMClient\lib
```

In our case, we copied `navigatorAPI.jar` to `C:\ICNLibs`, the local directory. The `navigatorAPI.jar` file will be referenced later in the plug-in project.

Now that the extensions are enabled for our Eclipse development environment, it is time to create the first IBM Content Navigator plug-in project. Proceed through the following steps:

1. Select **File** → **New** → **Other** and scroll until you find the `IBM Content Navigator` folder. Expand the folder and click **Content Navigator Plug-in**.
2. Click **Content Navigator PlugIn Project** and then click **Next**. A wizard opens, which guides you through the process of creating a Content Navigator plug-in as shown in Figure 3-3 on page 82.

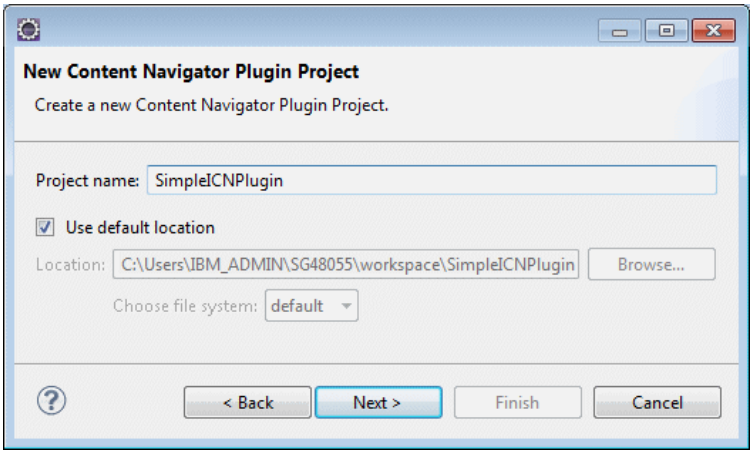



*Figure 3-3 New IBM Content Navigator plug-in project wizard*

3. Provide a name for your project, for example `SimpleICNPlugin`. Keep the default settings for the project location and click **Next**.
4. In the next window, define the values listed in Table 3-1. Figure 3-4 on page 83 shows the values provided for our SimplePlugin project. After setting all necessary parameters, click **Finish**.

*Table 3-1 Plug-in information in the new plug-in dialog*

| Plug-in information | Description |
|---|---|
| Descriptive Name | The descriptive name is the string that identifies your plug-in in the IBM Content Navigator administration plug-in interface. |
| Java Package | The Java package is the namespace for your plug-in source code. |
| Class Name | The class name is the name of your primary plug-in class. The primary plug-in class provides instructions to the IBM Content Navigator server about which extensions your plug-in provides and which classes to load at runtime. |
| Version | The Version number of the plug-in will be set in the Class provided under Class Name. |
| Location navigatorAPI.jar | The navigatorAPI.jar file contains the plug-in interfaces and classes provided by IBM Content Navigator that you will need to build your extensions. The navigatorAPI JAR file is installed with IBM Content Navigator under the `lib` directory (for example: `C:\Program Files\IBM\ECMClient\lib`). |

Figure 3-4 shows this information in the Eclipse development environment.

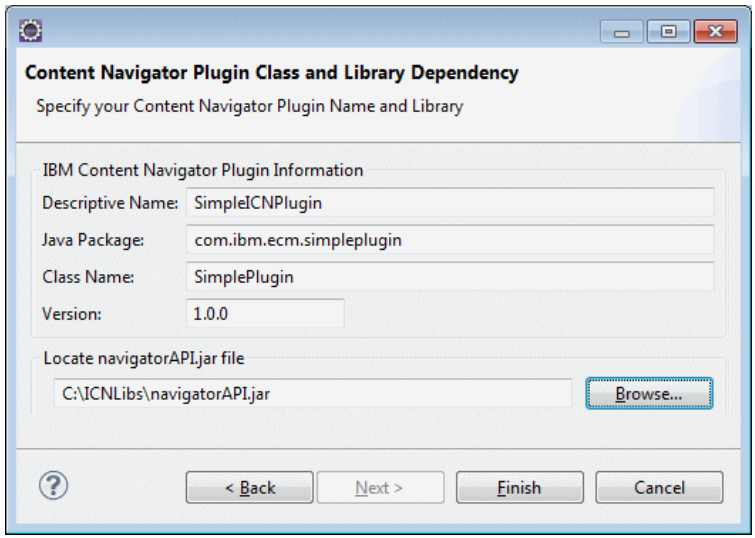



*Figure 3-4 Plug-in information in the new plug-in dialog*

The wizard generates a working Content Navigator plug-in project. You will see the primary plug-in class, with an ANT build script you can use to generate a deployable JAR file of your plug-in. Figure 3-5 demonstrates how the newly created plug-in project looks in the Eclipse package explorer.

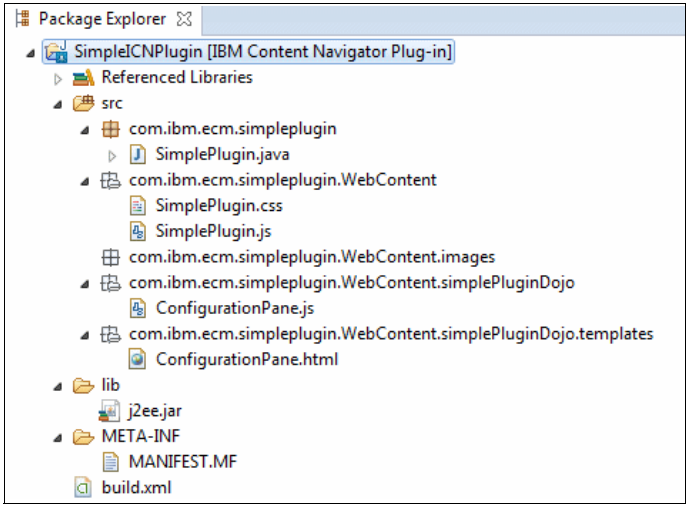



*Figure 3-5 Project structure of SimpleICNPlugin*

Before we continue with packaging and deploying of SimpleICNPlugin, we examine the created project structure.

As Figure 3-5 on page 83 shows, your project consists of a `src` folder, `lib` folder, and a `META-INF` folder. It also shows `build.xml`, the ANT build script file.

In addition to the primary plug-in class, notice that the wizard generates several other classes and packages within the project. Under the primary package name you provided in the plug-in project creation wizard, there is a `WebContent` directory. This directory is used as the root for any extensions your plug-in is providing to the client-side of IBM Content Navigator. In this case, the client-side extensions do not actually extend or change the user interface. They are shells you can use to build your customizations. For example, if you want to add a style change or override, you can add it to the CSS file generated under the `WebContent` directory. Also, if you want to prompt the administrator deploying your plug-in for custom configuration, you modify the ConfigurationPane JavaScript and HTML template generated by the wizard.

The wizard generates a Dojo package name for your custom Dojo widgets. This is the namespace that IBM Content Navigator registers for your Dojo widgets. For example, if we create a custom dialog, MyTestDialog, we add it to the simplePluginDojo package of this project and reference it as `simplePluginDojo.MyTestDialog`.

The `lib` folder contains the `j2ee.jar` file by default. To avoid major changes in the `build.xml`, you can add all JARs that are later required by your plug-in to the `lib` folder.

The `META-INF` folder contains a `MANIFEST.MF` file that defines the Main plug-in class for the plug-in project. In our project, the MANIFEST file parameter Plugin-Class points to `com.ibm.ecm.simpleplugin.SimplePlugin` and has automatically been set through the project creation wizard. The Plugin-Class parameter is used by the Content Navigator plug-in registration in the Administration Desktop to define the plug-in specification. If you build your project manually, it is important that the Plugin-Class points to the correct plug-in Java class.

The `build.xml` is an ANT script which compiles and generates a JAR file out of the existing project code. The `build.xml` file in our project is ready to run. Be aware if you extend the `SimpleICNPlugin` project, for example by adding actions, features, services, and so on, ensure that all necessary libraries (for example repository specific JARs) are referenced in the path section of the build script; otherwise, build errors will occur. In addition, the `build.xml` also contains the name of the output plug-in JAR file in the `jar` section. For details about the ANT script and packaging of the plug-in, see 3.3.3, “Packaging and building a plug-in” on page 90.

The SimpleICNProject we created here is an empty plug-in project. The following section shows how to add extensions to a Content Navigator plug-in project.

**Note:** Be sure you are using Java Compiler Level 1.6 in your plug-in project to avoid Java Version errors during plug-in registration.

## 3.3.2 Creating plug-in extension

The Content Navigator plug-in menu in Eclipse allows you to add extra plug-in components such as action, features, layouts, menus, viewer, and server extensions such as request and response filters to your plug-in.

Now that we have created a plug-in project and deployed it to IBM Content Navigator, it is time to create extensions for the plug-in. Right-click on the Java package in your plug-in project; you see a new menu named IBM Content Navigator. This menu type exposes new wizards for creating Content Navigator plug-in extensions. Several extensions are grouped into a submenu named Server Extensions, while others are located in the main menu. *Server extensions* are plug-in extensions that apply only to the Content Navigator server-side; other extensions can have client-side enhancements also. Figure 3-6 on page 86 shows what you see if you expand the **IBM Content Navigator** menu.

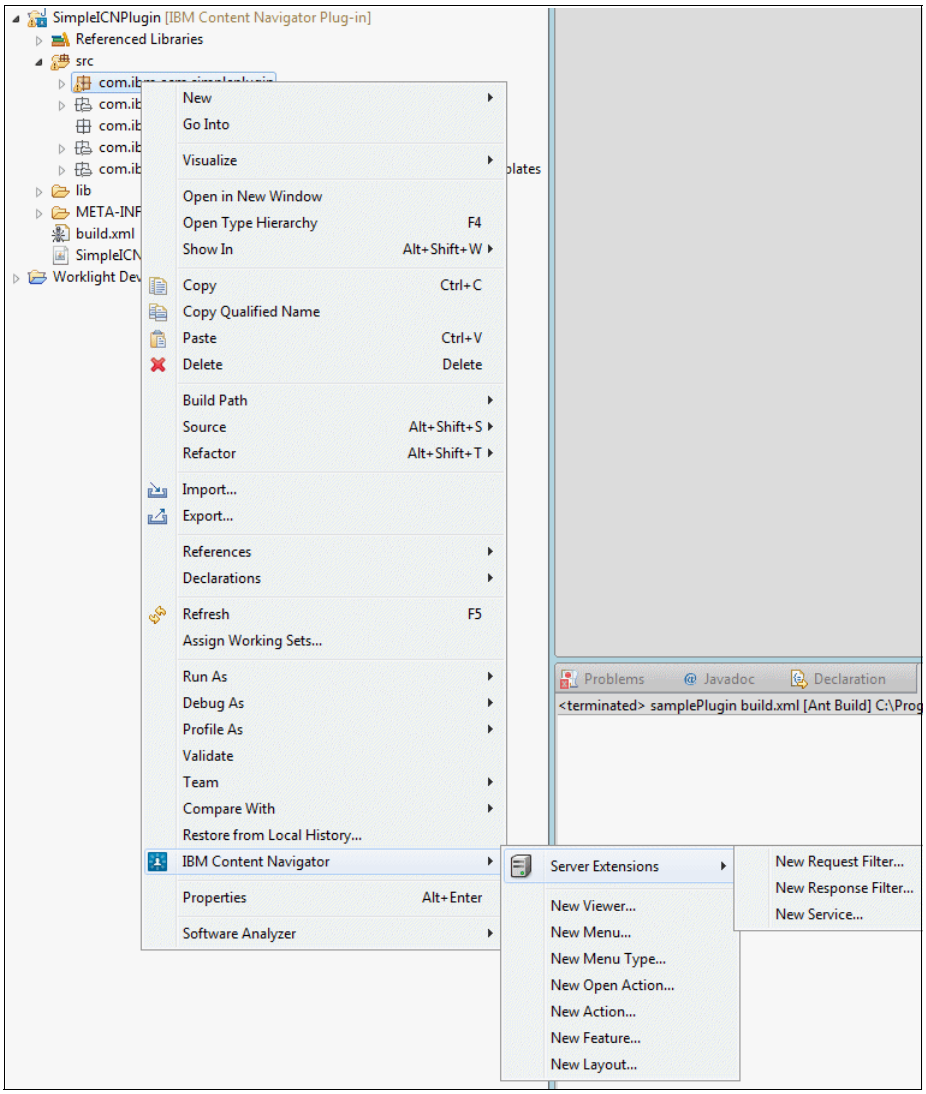



*Figure 3-6 Eclipse plug-in context menu*

The next section shows how new extensions can be added to a plug-in project.

## Server-side extensions

IBM Content Navigator provides several server-side extensions, including response filers, request filters, and services. If you want to add a response filter, select the `com.ibm.ecm.simpleplugin` java package, right-click and select **IBM Content Navigator** → **Server Extensions** → **New Response Filter**. *Response*

*filters* are extensions to existing IBM Content Navigator services, which allow you to manipulate the standard JavaScript Object Notation (JSON) payload returned by the server-side service. You can use this feature to insert custom formatter widgets in the list and property information user interface widgets, alter values within the JSON, or insert entirely new entries in the JSON return. Figure 3-7 shows a populated version of the New Response Filter wizard.

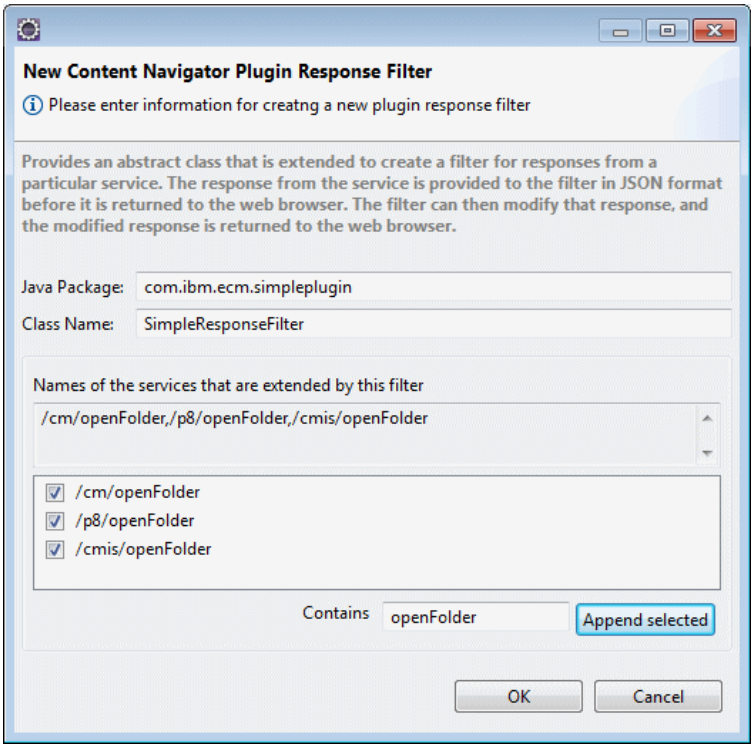



*Figure 3-7 New Response Filter wizard*

In this example, we searched for services responsible for handling openFolder actions and we added each service matching the search criteria to the response filter by selecting the appropriate check box and clicking **Append selected**. By adding each of the openFolder services to the list of services extended by this response filter, we are instructing IBM Content Navigator to call this response filter for any openFolder service request against an IBM Content Manager, IBM FileNet Content Manager, or Content Management Interoperability Services repository source. Click **OK**.

The wizard then generates the new response filter Java class and updates the primary plug-in Java class. The plug-in Java class update is necessary to instruct the IBM Content Navigator server that this plug-in contains a response filter. If

you enter the values specified previously, you see an update to your plug-in class as shown in Example 3-1.

*Example 3-1 Extension to plug-in class for Response Filter*

```
   /**
    * Provides a list of filters that are run after a requested
    * service. This list of filters can be used to modify the
    * the response that is returned.
    *
    * @return An array of
    *         <code>{@link com.ibm.ecm.extension.PluginResponseFilter
PluginResponseFilter}</code>
    *         objects.
    */
   public com.ibm.ecm.extension.PluginResponseFilter[]
getResponseFilters() {
      return new com.ibm.ecm.extension.PluginResponseFilter[]{new
com.ibm.ecm.simpleplugin.SimpleResponseFilter()};
   }
```

In addition to response filters, you can also create request filters and services. Request filters are similar in nature to response filters, but instead of applying after the IBM Content Navigator service has completed, a request filter is applied before the service executes. This allows the plug-in to manipulate the request parameters applied to the service prior to any action being taken by the service.

A plug-in service is used when you need to add an entirely new server-side procedure to IBM Content Navigator. This is the equivalent of building a custom servlet that provides some specific service. For example, you can use this method to expose additional capabilities from an ECM repository that are not yet available as ready-to-use.

## Client-side extensions

Now that a server-side extension is created, create an extension to the client-side. To open the IBM Content Navigator menu again, right-click the Java package, and click the **New Action** link. The wizard for creating an IBM Content Navigator action opens.

An *action* is an extension that allows the plug-in to define a new user interface action. Typically, actions are exposed as buttons in a toolbar or menu items in a context menu. However, they may also be actions executed implicitly during other user interactions within IBM Content Navigator. Figure 3-8 on page 89 shows a completed New Action wizard window.

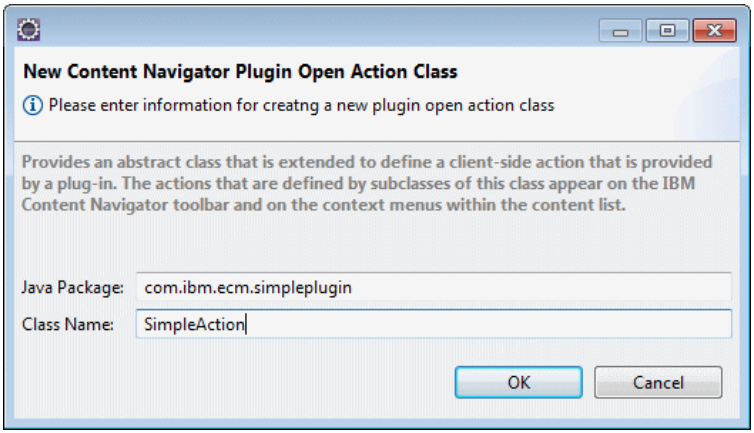



*Figure 3-8 New Action wizard*

After clicking **OK,** the wizard generates a new Java class for the action, updates the primary plugin Java class and updates the main plug-in JavaScript. The changes to the plug-in Java class are necessary to instruct the IBM Content Navigator server that this plug-in includes a custom action. If you enter the values specified above in Figure 3-8, you see the update to your primary plug-in class, as shown in Example 3-2.

*Example 3-2 Extension to plug-in class for Action*

```
   /**
    * Provides a list of actions that this plug-in adds to the main
    * toolbar of the web client.
    *
    * @return An array of
    *         <code>{@link com.ibm.ecm.extension.PluginAction
    * PluginAction}</code>
    *         objects. The plug-in should return the same set of
    * objects on every call.
    */
   public com.ibm.ecm.extension.PluginAction[] getActions() {
      return new com.ibm.ecm.extension.PluginAction[]{new
com.ibm.ecm.simpleplugin.SimpleAction()};
   }
```

This change allows your action to be called from anywhere within the user interface.

**Note:** Because the action method is defined globally, use a function name that is less likely to cause conflicts with other functions that are defined in the user interfaces. For example, you might choose to append your plug-in ID to every JavaScript function name; in this case, it would be SimplePlugin.simpleAction."

In addition to plug-in actions, you see wizards for creating viewers, menus, menu types, open actions, features and layouts. Table 3-2 briefly explains each plug-in type.

*Table 3-2 Plug-in types*

| Plug-in type | Description |
|---|---|
| Viewer | Plug-in viewers provide a mechanism for adding custom document viewers. |
| Menu | Menus allow a plug-in to define new menus of an existing menu type. For example, you can provide a custom version of the standard toolbar IBM Content Navigator displays above a list of content. |
| Menu Type | Menu types are entirely new menus not exposed by standard IBM Content Navigator widgets. Plug-ins can use this capability to define custom menus and toolbars for custom widgets added to the layout. |
| Open Action | An open action plug-in is a special type of action that applies only to the opening of documents. |
| Feature | Features are user interface panels that are exposed in the IBM Content Navigator standard layout widget. For example, the standard Search and Browse panels in IBM Content Navigator are features. Plug-in developers can add custom features that might add entirely new capabilities to IBM Content Navigator or extend existing features. |
| Layout | Layouts are used when you do not want to use the standard IBM Content Navigator layout, but instead want to provide a completely custom user experience. |

## 3.3.3 Packaging and building a plug-in

In 3.3.1, "Creating a simple plug-in project using the wizard" on page 81, we described how to create a simple plug-in project. To deploy a plug-in in Content Navigator, you must create an Java archive (JAR) file from the project code. In this section, we describe how to build a JAR file that later can be installed in IBM Content Navigator. Before you start building the JAR file, ensure that no compilation errors exist in your simple plug-in project.

Two ways are available to build a JAR file from the plug-in project:

- Export the project using the default export functionality of Eclipse and Rational Application Developer (see 3.6, “Building a plug-in JAR manually” on page 109).
- Use an ANT script to build the JAR.

Our SimpleICNPlugin project already contains a working `build.xml` file, which can be used to create the JAR file. We provide details about this ANT script.

We focus on two major sections of the `build.xml`:

- The first section contains the class path tag, which must include all libraries that are necessary in order to compile your plug-in project.
- The second section (JAR section) contains the Main plug-in class of your project.

## Description of class path section

The class path section contains all necessary libraries to compile the plug-in project. In our case, it is necessary to have only the `navigatorAPI.jar` and the `j2ee.jar` to compile the project. The Content Navigator Plug-in wizard created the entries of the `build.xml` from the location of the provided `navigatorAPI.jar` (see Figure 3-4 on page 83) and also for the `j2ee.jar` automatically. The `temp` folder will contain the compiled class files created through the ANT script and is also created automatically.

You do not need to change the paths unless you need special libraries in your plug-in code. In this case, add an extra path element in the class path section. Example 3-3 shows the `build.xml` file generated automatically.

*Example 3-3 Class path section of build.xml for SimpleICNPlugin*

```
<path id="class path">
      <pathelement location="C:/ICNLibs/navigatorAPI.jar" />
      <pathelement location="./lib/j2ee.jar" />
      <pathelement location="./temp" />
</path>
```

## Description of JAR section

The JAR section of `build.xml` creates the Manifest file for the plug-in. It is important that the parameter for the attribute Plugin-Class matches the Java Class name that extends the `com.ibm.ecm.extension.Plugin` class. In addition, the name of the JAR file to be generated can be set in this section.

*Example 3-4 The jar section of build.xml for SimpleICNPlugin*

```
<jar jarfile="SimpleICNPlugin.jar">
        <fileset dir="./temp" includes="**/*" />
        <manifest>
           <attribute name="Plugin-Class"
value="com.ibm.ecm.simpleplugin.SimplePlugin" />
           <section name="build">
              <attribute name="Built-By" value="${user.name}" />
              <attribute name="Build" value="${TODAY}" />
           </section>
        </manifest>
</jar>
```

## Execute the build

To build the plug-in JAR file, complete the following steps:

1. Open the `build.xml` file within your development environment.
2. Within Eclipse, find the special Run configuration for ANT scripts. Right-click the **build.xml** file and select **Run as** → **Ant Build**. See Figure 3-9 on page 93.

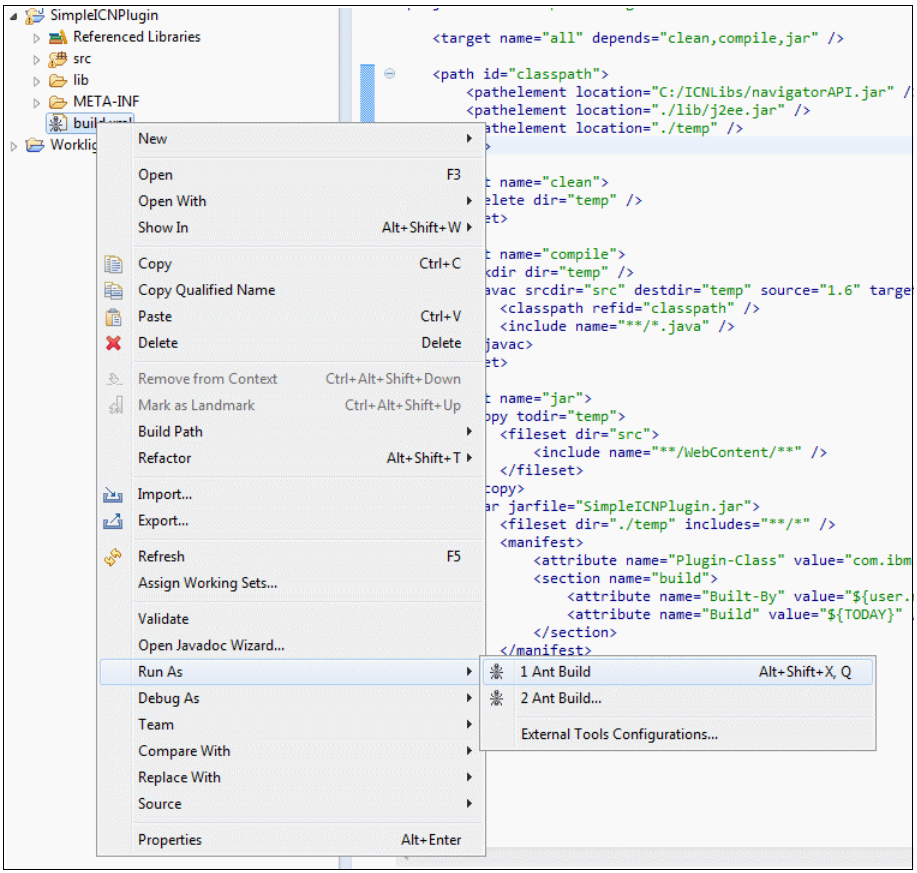



*Figure 3-9 Execute ANT script*

3. The Eclipse Console view shows the execution of the build. The build ends with a build successful statement. If it does not, inspect the error message and make the necessary changes.
4. Refresh your project. Within the project, you see the `SimpleICNPlugin.jar` file.

**Note:** If you want to build the JAR manually, see 3.6, “Building a plug-in JAR manually” on page 109.

## 3.3.4 Registering and testing a plug-in

This section explains how to register a plug-in JAR file in IBM Content Navigator.

IBM Content Navigator provides two mechanisms for loading plug-ins:

- The first mechanism requires you to build your plug-in as a JAR file and specify a path to the JAR file. This path can be a local file system path or a URL path. For production deployments, build your plug-in as a JAR file and load it through a URL path.
- To simplify your development efforts, a second option allows you to provide the file system path to your plug-in project "bin" directory and the full class name of your primary plug-in class. Using this mechanism allows you to make changes to the plug-in source code and test it immediately within IBM Content Navigator, without having to first rebuild and reload a plug-in JAR file. This reduces the steps required to develop and test a new plug-in.

The next section describes how to register a plug-in as class file path and name.

### Test a plug-in within development lifecycle

With IBM Content Navigator 2.0.2, it is possible to test a plug-in within the development lifecycle and without the need to package the JAR file during the development process. In this section, we describe what is necessary to use this feature and we demonstrate the use with our example project.

First, the project we created must be accessible on the Content Navigator Server. Assuming your development environment is on your local workstation, you should have a workspace on the Content Navigator server file system, because the plug-in registration can handle only server-side file systems. Therefore, you can map a network drive from your local workstation to the Content Navigator server. Another option is to copy the project to the Content Navigator server. If you develop directly on the Content Navigator server, which means your development environment is installed on the Content Navigator server, you do not need to handle the network drives or connections.

We decided to connect our local workstation development environment to a workspace on the Content Navigator server so the project can be accessed directly. Our workspace is located in `C:\workspace` on the Content Navigator Server.

To register the plug-in without doing the packaging, use these steps:

1. Open the Administration Desktop and go to the plug-ins tab, which lists all the plug-ins that are already installed in your environment. See Figure 3-10.

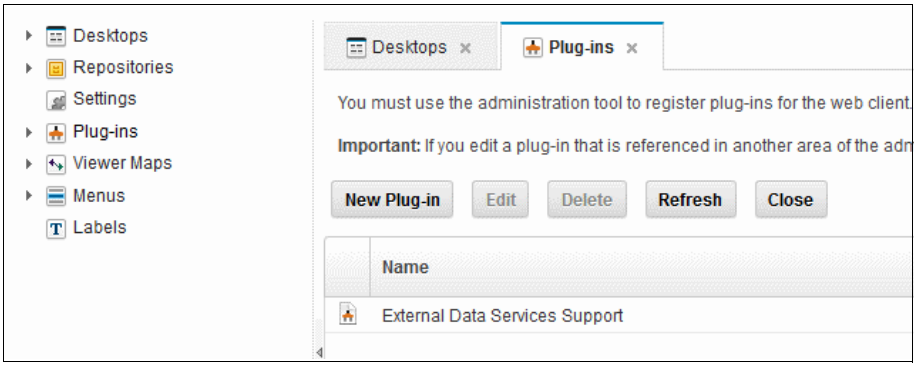



*Figure 3-10 Installed plug-ins in IBM Content Navigator*

2. Click **New Plug-in** and specify the path to your plug-in.
3. Select the **Class file path** radio button and provide the path to your `bin` folder within your project in the workspace. In the **Class name** field, provide the name of the plug-in class file including the whole Java package. The class file path in our case is `C:\workspace\SimpleICNPlugin\bin` and the name of the class file is `com.ibm.ecm.simpleplugin.SimplePlugin`.
4. Click **Load**. The plug-in is loaded, as shown in Figure 3-11 on page 96.

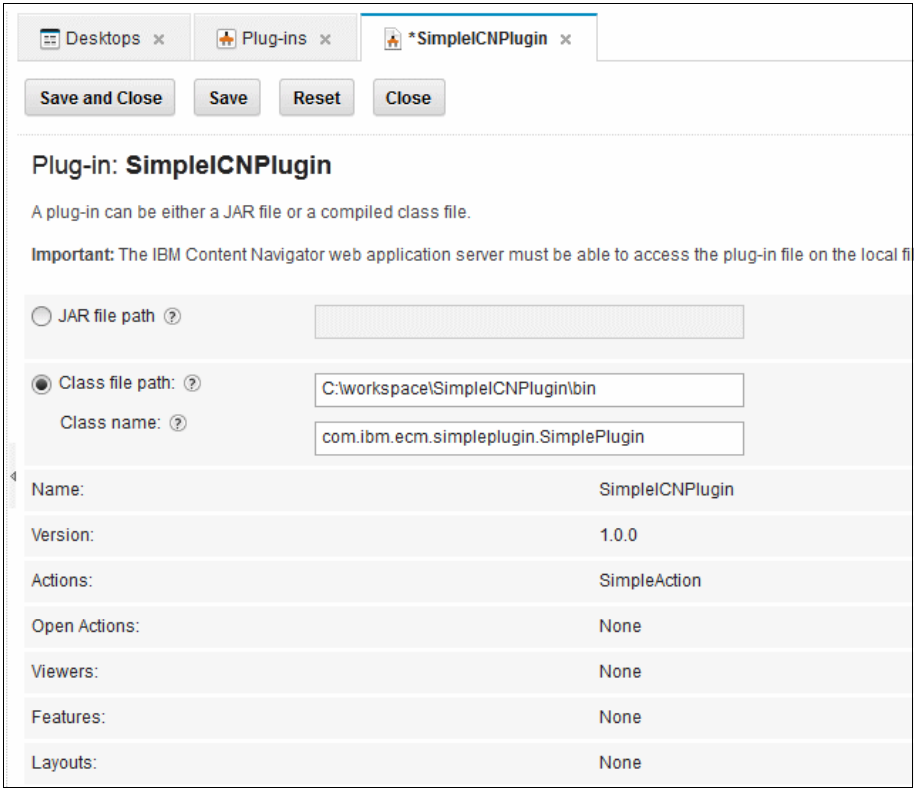



*Figure 3-11 Plug-in registration by class file*

5. Click **Save** and **Close**.

## Register a plug-in JAR file

For registration of the JAR file, we use the `SimpleICNPlugin.jar` file that was created in 3.3.3, “Packaging and building a plug-in” on page 90.

The plug-in JAR file must be in a location where the IBM Content Navigator application has access. This access is especially important for high availability scenarios. If the plug-in JAR file is not available at the defined location or is erroneous, it can not be registered or executed correctly. During plug-in registration, you can choose between providing a file path to the JAR file or using a URL path to the JAR.

Follow the steps to deploy the plug-in created in the previous section:

1. Open the Administration Desktop and click the **Plug ins** tab, which lists all plug-ins that are already installed in your environment (Figure 3-12).

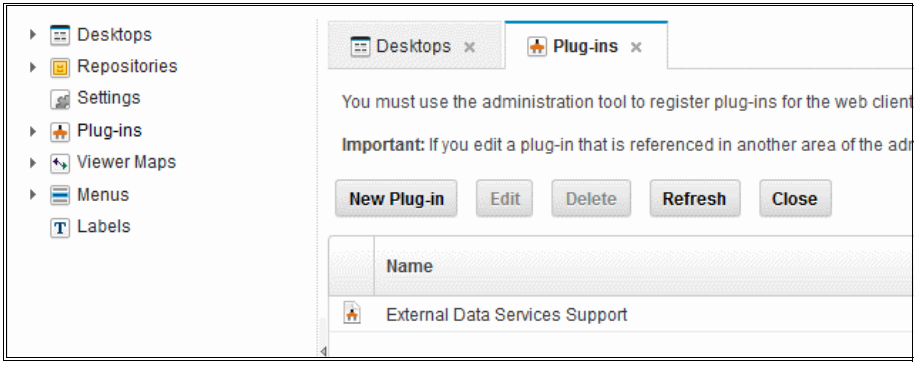



*Figure 3-12 Installed plug-ins in IBM Content Navigator*

2. Click **New Plug-in** and specify the path to your plug-in. Select the **JAR file path** radio button (Figure 3-13) and provide the full path to the plug-in JAR file. In our case, the file location is as follows:

   `C:\ICNPlugins\SimpleICNPlugin.jar`

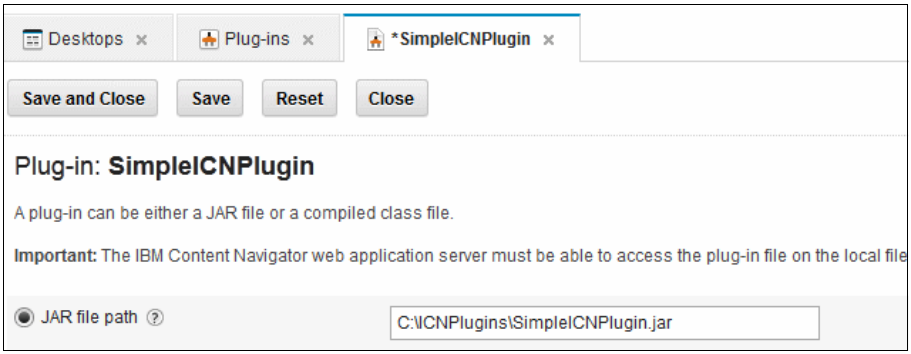



*Figure 3-13 Publish a new plug-in within Administration Desktop*

3. Click **Load**.

   If the plug-in is built correctly, information about the plug-in is displayed, for example, the version number, the name, available actions, viewers, features, and layouts. In our case, we have only a simple plug-in without any functions.

4. Click **Save** and **Close** to save and close the plug-in.

Now the plug-in is displayed in the plug-ins view.

## Considerations for plug-in deployment in production environments

At the end of every development cycle, you deploy the plug-in in various stages, starting in a test environment, going to the user acceptance test environment, and ending at the production environment. Most companies have a well-defined deployment process of how new pieces of software must be delivered and deployed through all of these stages. From the Content Navigator perspective, often more than one single plug-in JAR file is to be deployed; that is, multiple components developed by different people or departments. You will also find company-specific resources that must be embedded and addressable through a URL in the Administrator desktop. To provide a consistent deployment process, we suggest you package all extensions, plug-ins, logos, logon sites, and so on, in one web application that can be versioned and deployed through all environments through a standard deployment procedure.

Figure 3-14 on page 99 shows four stages: development, system integration, user acceptance, and production environment. In the first stage, the developer usually creates new Content Navigator plug-ins and test the extensions in developer's owned environment. After development and testing is complete, the created plug-in and other custom components, such as logos, are packaged to one web application, which then can be deployed through all the stages, from system integration to production. The additional resources contained in the web application, for example logos, can than be used for desktop customizations.

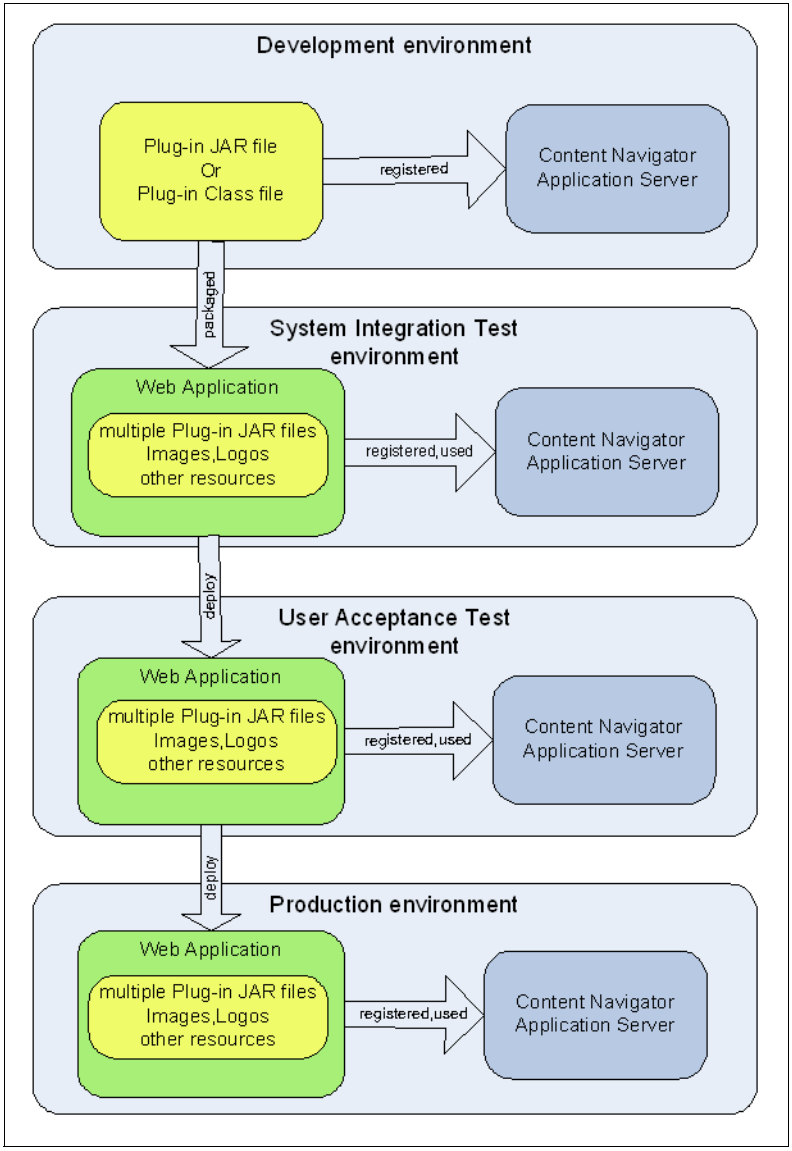



*Figure 3-14 Deploy extension through different stages*

This section describes how to create a new web application for customization, how to add the new plug-in to this web application, and how the plug-in is deployed in IBM Content Navigator. You may also reuse an existing web application for this purpose; however, to ease the maintenance efforts, using a dedicated web application is recommended.

To create a new web application in your development environment, complete the following steps:

1. Use *one* of the following steps, depending on how you want to deploy the web project in your environment:
   - Select **File** → **New** → **Dynamic Web Project**.
   - Select **File** → **New** → **Static Web Project**.

   If you choose a static web project, you can define the application context during the deployment and add this static web project to an existing EAR file. We select a Dynamic Web Project for this task.
2. Provide a name for your project, for example, customConfig. Define the runtime environment, select the **Add project to EAR** box, and provide a name for the EAR file. If you want to add this project to an existing EAR, select the name of the EAR project from the drop-down list. The runtime environment selected must match the server where you want the web application to deploy. Click **Finish**. If you want to install the web application in a WebLogic server, select the appropriate target runtime and configuration.
3. Open the customConfig project and add the `SimpleICNPlugin.jar` (that was created in 3.3.3, “Packaging and building a plug-in” on page 90) to the `WebContent` folder.
4. The application can now be deployed to your application server, for instance by using the Administration Console of WebSphere. Complete these steps:
   a. Export the customConfigEAR project as an EAR file to a user-defined location. Right-click your project, select **Export** → **EAR file**, and provide the necessary parameters, such as export destination.
   b. Deploy your EAR to your Application Server instance. For details see the appropriate application server documentation.
5. Make sure you can reach the web application and the JAR file through the URL. The URL depends on the server where you deployed the web application and on the context you provided. If you follow our example, the context is customConfig. Assuming the web application is deployed on a local server (localhost) and in a default WebSphere Application Server instance, the URL will be similar to the following example:

   ```
   http://localhost:9080/customConfig/SimpleICNPlugin.jar
   ```

   A window opens where you specify whether you want to save the JAR file.
6. To register the plug-in, complete the following steps:
   a. Open the Administration Desktop, click the **Plug-ins** tab, which lists all the plug-ins that are already installed in your environment.
   b. Click **New Plug-in** and specify the path to your plug-in. Enter the URL that references to your `SimpleICNPlugin.jar`. Our example is as follows:

```
http://localhost:9080/customConfig/SimpleICNPlugin.jar
```

c. After specifying the URL path to the JAR file, click **Load**.

   If the plug-in is built correctly, information about the plug-in is displayed, for example, the version number that the plug-in has, available actions, viewers, features, and layouts.

d. Click **Save** and **Close**. Now, the plug-in is displayed in the plug-ins view.

## 3.4 Creating a new empty EDS project using the wizard

In addition to the Eclipse tools for creating an IBM Content Navigator plug-in, you will also find a new web facet available to assist in the creation of an external data service. An external data service is a REST API, called by the IBM Content Navigator external data service plug-in, which uses data from an external source, such as a file or data in a database, to customize field properties and manage property behavior. The Eclipse tooling generates the external data service APIs as Java Servlets. However, you can implement the service in other programming languages, for example a .NET service, or as a JAX-RS API.

During the next steps, you create an empty EDS Web Application in your development environment, deploy the application, and register and configure the EDS plug-in.

To create a project using the Eclipse tooling, switch to the Java EE perspective and follow these steps:

1. Click **File** → **New** → **Project** → **Web** → **Dynamic Web Project**.
2. Eclipse opens the standard dynamic web project wizard (Figure 3-15 on page 102). On the first panel of the wizard, click the **Configuration** drop-down and select I**BM External Data Service**.

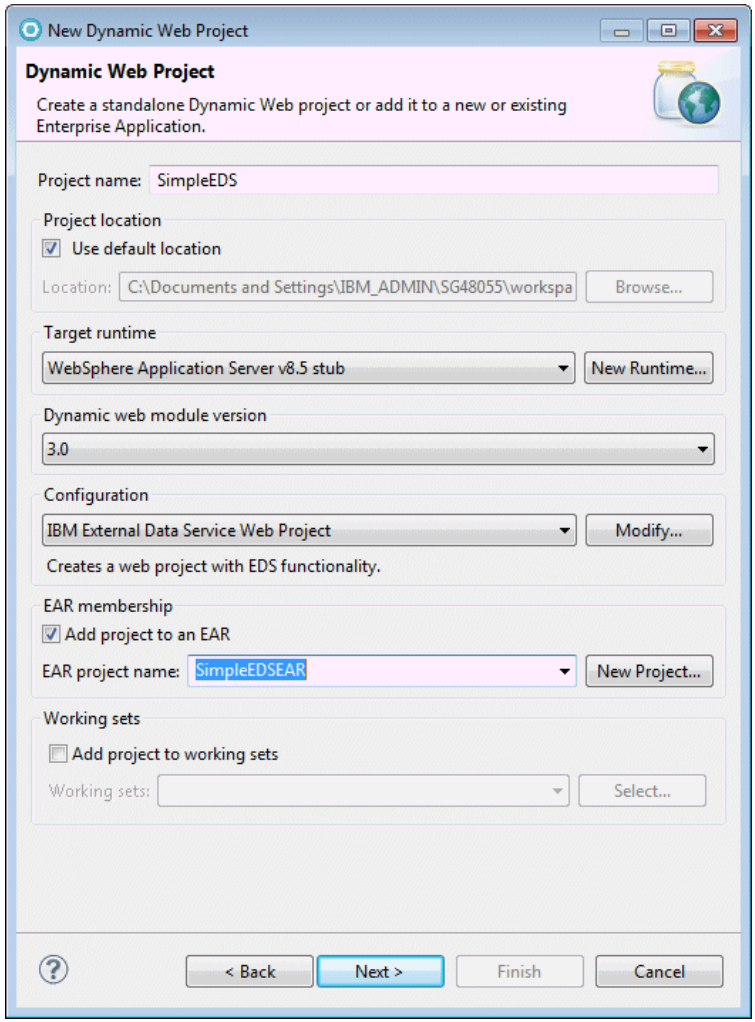



*Figure 3-15 New EDS application wizard*

3. Provide a name for the project and click **Next**.
4. The next window provides an optional dialog for adding folders to the web project source folder. After you add folders, if any, click **Next** to move to the next panel. On this panel (Figure 3-16 on page 103), you have the option to change the context root and content directory of the web project. You can also optionally generate a web deployment descriptor.

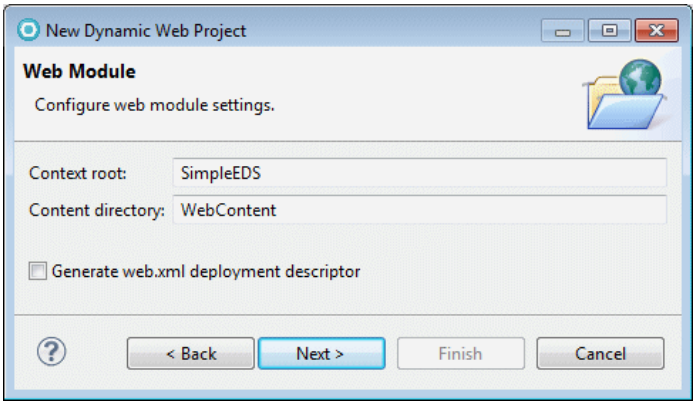



*Figure 3-16 Context root for EDS implementation*

5. Click **Next**.
6. The next window (Figure 3-17 on page 104) asks for a Java package name for your default external data service servlet classes and also the location of your `navigatorAPI.jar` file. As mentioned in the previous sections, we copied the `navigatorAPI.jar` from our Content Navigator server to a local directory named `C:\ICNLibs`. For this sample, we set the package name to `com.ibm.ecm.edsimpl`.

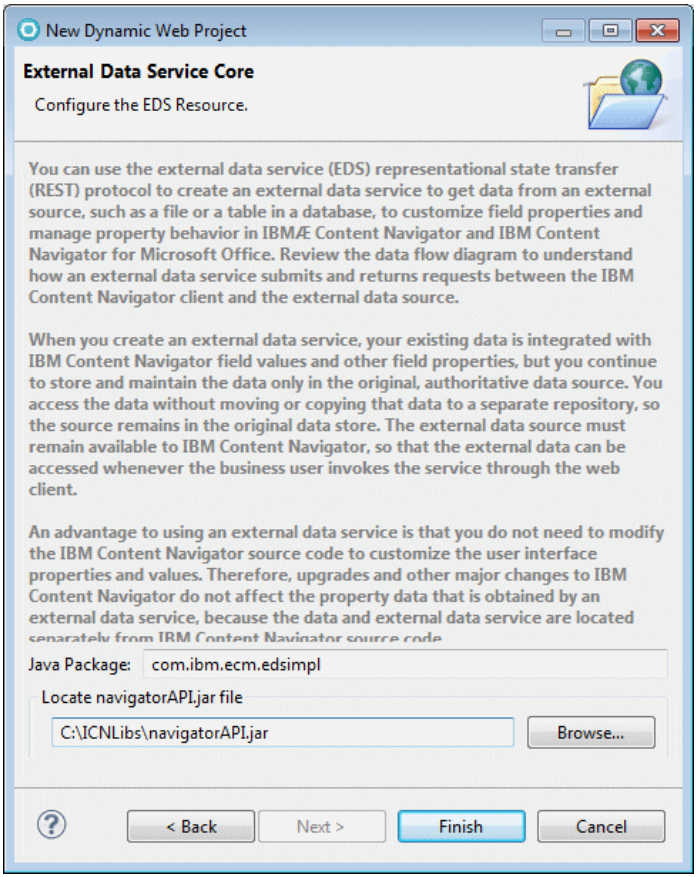



*Figure 3-17 Java package declaration for EDS implementation*

7. Click **Finish**.

The new project is being created. You see two servlets were created.

- The first, GetObjectTypesServlet, is the API called by the IBM Content Navigator external data service plug-in to determine the object types that this external data service project is augmenting. Example 3-5 shows a return value.

*Example 3-5 Sample return value*

```
[
{"symbolicName" : "Document"}
]
```

In this example, the EDS service instructed the EDS plug-in in IBM Content Navigator that it should be called any time the user accesses the Document class.

- The second servlet, UpdateObjectTypeServlet, is the API that the EDS plug-in will call when the user accesses an object type specified by the GetObjectTypesServlet. For example, this service might return a JSON payload that instructs IBM Content Navigator to set a property on the Document class to read-only. It might also hide properties, pre-populate properties, create choice lists and choice list dependencies, and so on.

Chapter 7, “Implementing request and response filters and external data services” on page 233 describes the implementation of an external data service.

After implementing the two servlets according to your requirements, the web application must be deployed in an application server (for example WebSphere Application Server). The EDS plug-in, delivered with the product, must be registered and configured to point to your EDS application. Follow these steps:

1. Export the project as an EAR file:
   a. Right-click the SimpleEDSEAR application and select **Export** → **EAR file**.
   b. In the export dialog, define a location where the EAR file should be exported to, for example `C:\SimpleEDSEAR.ear`.
   c. Click **Finish**.
2. Deploy the EAR file on WebSphere Application Server:
   a. Open and log in to the WebSphere Application Server administration console.
   b. Select **Applications** → **New Application** → **New Enterprise Application**.
   c. Select the installation path to the EAR file that you exported in step 1 and then click **Next** to go through all the steps. Click **Finish**, and then click **Save** to save the master configuration.
   d. Start the application if it has not been started yet.
   e. Test the application by entering the following URL. The context SimpleEDS can be set either within the project itself or during deployment of the application. The URL for our sample is as follows:

      ```
      http://localhost:9080/SimpleEDS/types
      ```

      The output of this URL should not contain any errors, only an empty page.

3. Register the EDS plug-in and configure EDS:
   a. Open the Content Navigator Administration Desktop.
   b. Go to **Plug-ins** and select **New Plug-in**.
   c. Select the **JAR file path** radio button and provide the URL to the `edsPlugin.jar` file in the Content Navigator installation path, as in the following example:

      `C:\Program Files (x86)\IBM\ECMClient\plugins\edsPlugin.jar`

      In the configuration field at the bottom, provide the URL for the application we deployed in step 2.
   d. Click **Save** and **Close**.

## 3.5 Getting started with the SamplePlugin

IBM Content Navigator provides a sample plug-in that contains samples for basic functions like usage of actions, filters, features, viewers, menus, and services. The SamplePlugin source code is located in the installation directory of IBM Content Navigator:

`<`*`CONTENT_NAVIGATOR_PATH`*`>\samples\samplePlugin`

You can import the sample plug-in into your development environment by selecting **File** → **Import** → **Existing Projects into Workspace**. Select the samplePlugin from your Content Navigator installation and the project will be imported.

The imported project has some class path errors you must resolve, if you want to rebuild and redeploy. Therefore changing the projects class path and the `build.xml` file is necessary.

Table 3-3 lists the necessary JAR files for compiling the SamplePlugin and their locations.

*Table 3-3 JAR files required for the SamplePlugin project*

| Name of JAR file | Location of JAR file |
|---|---|
| navigatorAPI.jar | <*ICN_INSTALL_PATH*>\ECMClient\lib |
| j2ee.jar | <*WAS_INSTALL_PATH*>\AppServer\lib |
| struts-1.1.jar | <*CN_DEPLOYED_PATH*>\navigator.war\WEB-INF\lib |
| commons-lang-2.3.jar | <*CN_DEPLOYED_PATH*>\navigator.war\WEB-INF\lib |
| commons-codec-1.4.jar | <*CN_DEPLOYED_PATH*>\navigator.war\WEB-INF\lib |
| Web-INF classes folder | This folder is necessary only if you want to compile the class SamplePluginGetAnnotationsService; you must involve the following path: <*CN_DEPLOYED_PATH*>\navigator.war\WEB-INF\classes |
| **IBM FileNet P8 Library** | |
| Jace.jar | <*CN_DEPLOYED_PATH*>\navigator.war\WEB-INF\lib |
| **IBM Content Manager OnDemand Library** | |
| ODApi.jar | ODWEK: <*ODWEK_INSTALL_PATH*>\api |
| **IBM Content Manager Library** | |
| cmbicmsdk81.jar | <*CN_DEPLOYED_PATH*>\navigator.war\WEB-INF\lib |
| **CMIS Libraries** | |
| chemistry-opencmis-client-api-0.8.0.jar | <*CN_DEPLOYED_PATH*>\navigator.war\WEB-INF\lib |
| chemistry-opencmis-client-bindings-0.8.0.jar | <*CN_DEPLOYED_PATH*>\navigator.war\WEB-INF\lib |
| chemistry-opencmis-client-impl-0.8.0.jar | <*CN_DEPLOYED_PATH*>\navigator.war\WEB-INF\lib |
| chemistry-opencmis-client-commons-api-0.8.0.jar | <*CN_DEPLOYED_PATH*>\navigator.war\WEB-INF\lib |
| chemistry-opencmis-client-commons-impl-0.8.0.jar | <*CN_DEPLOYED_PATH*>\navigator.war\WEB-INF\lib |

Besides the project's build path, you must also change the `build.xml` of the samplePlugin. The `build.xml` file must contain the location of the JAR files in order to run and compile correctly. Therefore, ensure that the class path plug-in section of the `build.xml` contains all files listed above. We did this through changing the property values for `J2EE_HOME`, `NAVIGATOR_HOME`, `ODWEK_HOME`,`CM_HOME`, and `P8_HOME`, as shown in Example 3-6.

We point the CM_HOME and the P8_HOME location to the directory where Content Navigator was installed in WebSphere Application Server. There you can find all the necessary JARs for the repositories. If you installed the repository APIs on your development workstation, you can also point to their library folders. The `ODApi.jar` is not included in the navigator deployment, so ODWEK must be installed and referenced directly to compile also the Content Manager onDemand samples.

*Example 3-6 Sample Plugin build.xml changes*

```
<property name="NAVIGATOR_HOME" value="C:/Program Files
(x86)/IBM/WebSphere/AppServer/profiles/AppSrv01/installedApps/icntraini
ngNode01Cell/navigator.ear/navigator.war" />

<property name="J2EE_HOME" value="C:/Program Files
(x86)/IBM/WebSphere/AppServer/lib" />

<property name="P8_HOME" value="${NAVIGATOR_HOME}" />

<property name="ODWEK_HOME" value="C:/Program Files/IBM/OnDemand Web
Enablement Kit/api" />

<property name="CM_HOME" value="${NAVIGATOR_HOME}" />
```

After changing the `build.xml` accordingly, you can build the JAR file as described in "Execute the build" on page 92. The `SamplePlugin.jar` is generated and can now be deployed in your Content Navigator environment. See "Register a plug-in JAR file" on page 96 for the `SamplePlugin.jar` file. You can also use the JAR that is delivered within the Content Navigator installation package:

`C:\Program Files (x86)\IBM\ECMClient\plugins\SamplePlugin.jar`

## 3.6 Building a plug-in JAR manually

Building a plug-in JAR file manually can be useful, if you have included all the necessary JARs directly in your projects build path and you do not want to change the `build.xml` file to contain these JARs.

Complete the following steps to manually generate a JAR file by using Eclipse or Rational Application Developer:

1. Create a manifest file that specifies the Java plug-in class. Example 3-7 shows the manifest file for JAR file generation. An important aspect about the manifest file is the value for the attribute *Plugin-Class*. This value must correspond to your main Java plug-in class, which extends the IBM Content Navigator plug-in class.

   *Example 3-7 Manifest file for JAR file generation*

   ```
   Manifest-Version: 1.0
   Ant-Version: Apache Ant 1.7.1
   Created-By: 2.4 (IBM Corporation)
   Plugin-Class: com.ibm.ecm.simpleplugin.SimplePlugin
   Name: build
   Built-By: icnPlgin
   Build: ${TODAY}
   ```

2. Export the JAR file:
   a. Right-click the project and select **Export** → **Java/JAR File**. The Java Project file selection window opens.
   b. Deselect the `lib` folder, and specify a location and a file name for your plug-in JAR file. The `lib` folder does not need to be included in the target JAR file because the containing JAR files are necessary only to compile the project and are included in the Content Navigator application.
   c. Click **Next**.
3. The Java Packaging Options window opens. Keep the default options and click **Next**.
4. The JAR Manifest Specification window opens. Select **Use existing manifest from workspace** and provide the path to the manifest file you created before. Click **Finish**.

The plug-in project is now exported as a JAR file, by using Eclipse or Rational Application Developer

# 3.7 Debugging plug-ins

This section provides an overview of how to enable debugging for the Java part of plug-ins in your development environment. For more information regarding troubleshooting and debugging, see Chapter 14, “Debugging and troubleshooting” on page 495.

Debugging plug-ins can be useful to determine the cause of certain issues regarding the Java part of a plug-in. You can either enable remote debugging or debug your applications locally.

## 3.7.1 Remote debugging

To debug your application remotely, turn on debugging for the WebSphere Application Server at the Administration console and enable your development environment to access the dedicated application server.

The following steps describe the process to enable remote debugging for WebSphere Application Server and Rational Application Developer:

1. Enable debugging on your WebSphere Application Server where Content Navigator is installed:
   a. Open the Administration console and log in:

      `http://<name_of_host>:<port>/ibm/console`

      For example:

      `http://localhost:9060/ibm/console`

   b. Go to **Application Servers** → ***<name_of_your_appServer>*** → **Additional properties** → **Debugging service**.

   c. Select the **Enable service at server startup** check box.

   d. Remember the port number shown in Figure 3-18 on page 111. This port number is necessary for configuring the Rational Application Developer environment later. Make sure that this port is not used by any other application.

   e. Click **Apply** and **Save** to save the changes that you made and exit the Administration console.

   f. Restart the WebSphere Application Server.

   By the time server is started, the debugging port is available.

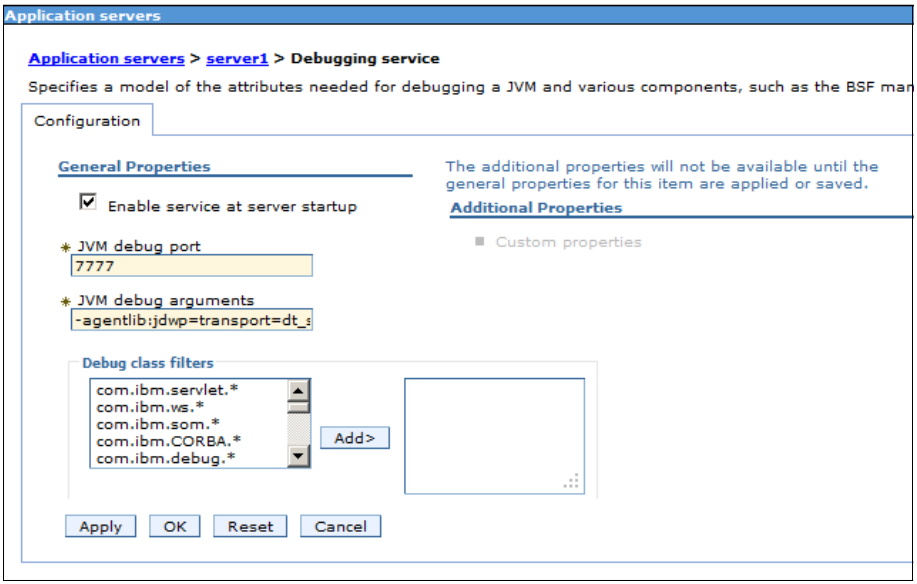



*Figure 3-18 Enable debugging on WebSphere Administration Console*

2. Add a new Runtime Configuration in Rational Application Developer:

   a. Go to **Run** → **Debug Configurations**.

   b. Select **Remote Java Application** from the list and click **New configuration** at the upper left corner.

   c. Enter a new name for the configuration, for example, `DebugSamplePlugin`.

   d. Enter the name of the project you want to debug or select it from the **Browse** tree. For our example, we select the SamplePlugin project.

   e. Enter the host name of your WebSphere Application Server and enter the port number from step 1b on page 110. Determine whether the host name is available and can be accessed from your development environment.

   f. Click **Apply** and **Close** to save the configuration.

   A sample of the configuration is displayed in Figure 3-19 on page 112.

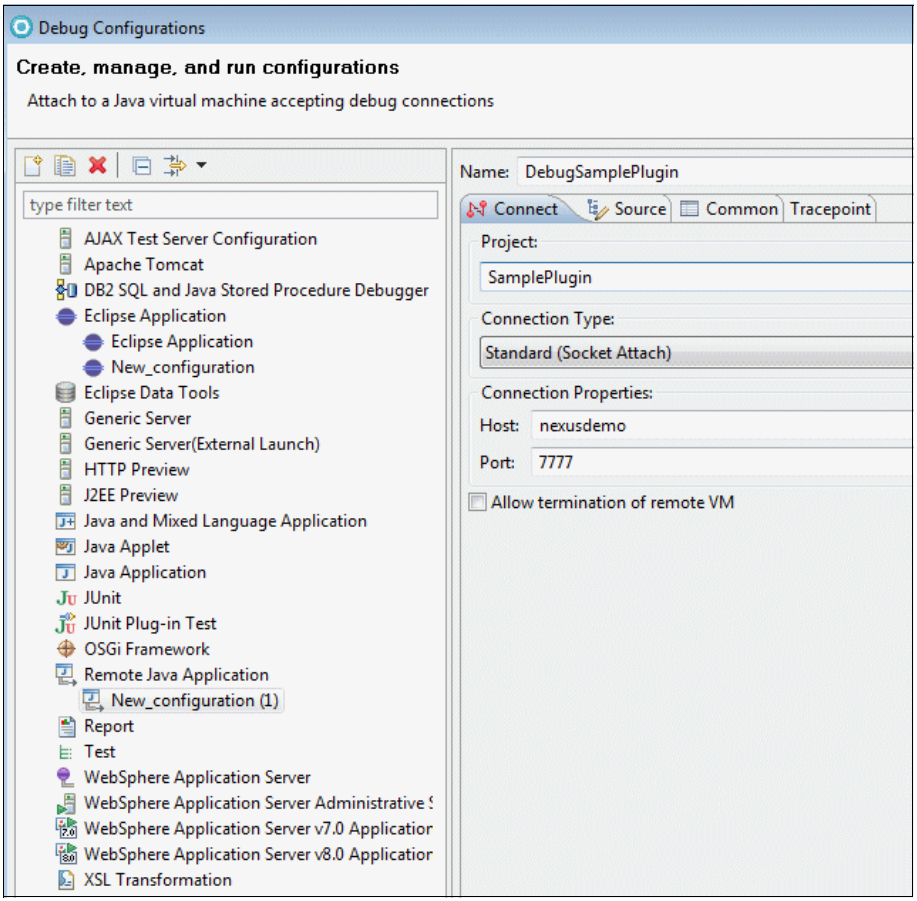



*Figure 3-19 Create new debug configuration: Set connect information*

3. Add breakpoints in your source code.
4. Start debugging by selecting **Debug** → **Debug configuration** → **DebugSamplePlugin** → **Debug.**
5. Start your Content Navigator in a web browser. When the web application reaches a breakpoint, you see it in the development environment.

### 3.7.2 Local debugging

If you run the Content Navigator and the development environment on the same physical machine, follow these steps to enable debugging for the server:

1. Set breakpoints in your Java code.
2. Open the Server view in Rational Application Developer by clicking **Window** → **Open View** → **Server**.
3. Select the server where Content Navigator is installed, right-click the **Server**, and select **Restart in Debug** mode.
4. Start your Content Navigator in the web browser. When the web application reaches a breakpoint, you see it in the development environment.

For more information about application debugging, see the IBM Knowledge Center for both WebSphere and the Rational Application Developer.

## 3.8 Conclusion

This chapter describes how to set up the development environment for IBM Content Navigator. It offers suggestions and recommendations for deploying a new plug-in. The remaining chapters in this book provides concrete use cases of customizing and extending IBM Content Navigator with sample code.

# Part 2

# Customizing Content Navigator

In this part, we show how to customize and extend Content Navigator with multiple code samples. The code samples include how to develop actions, services, features, request filter, EDS, custom step, mobile application. We also show how to work with built-in viewers, integrate third-party viewers, and more. This part contains the following chapters:


- Chapter 4, “Developing a plug-in with basic extension points” on page 117
- Chapter 5, “Building a custom repository search service” on page 173
- Chapter 6, “Creating a feature with search services and widgets” on page 211
- Chapter 7, “Implementing request and response filters and external data services” on page 233
- Chapter 8, “Creating a custom step processor” on page 277
- Chapter 9, “Using Content Navigator widgets in other applications” on page 297
- Chapter 10, “Customizing built-in viewers and integrating third-party viewers” on page 365
- Chapter 11, “Extending solutions to mobile platform” on page 413
- Chapter 12, “Extending Profile Plug-in for Microsoft Lync Server” on page 437


© Copyright IBM Corp. 2012, 2014. All rights reserved.

4

# Developing a plug-in with basic extension points

This chapter describes how to create an IBM Content Navigator plug-in to customize and extend its base functionality. The chapter first provides information about the examples we will be implementing and then describes how to implement the IBM Content Navigator plug-in.

This chapter covers the following topics:

- Example overview
- Developing the Create Dossier action
- Developing a plug-in service to create the substructure of the dossier
- Developing the Open Dossier action
- Open dossier view in its own feature
- Adding configuration
- Dossier management in the real world

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# 4.1 Example overview

In the next sections we introduce examples of how to extend IBM Content Navigator at the various extension points introduced in Chapter 1, “Extension points and customization options” on page 3.

To make the examples more illustrative, we introduce the requirements of Redbooks Fictional Company A that wants to customize the IBM Content Navigator to meet the needs of its workers. This fictitious company A wants to organize the electronic documents of its customers in a way that is analogous to how it was done with the physical binders and files when paper was used.

There is one single physical binder for all documents of a single customer and the documents are structured by document types into different slots. A *dossier* is an electronic equivalent to the physical binder. Technically, it is a top-level folder (with a specific folder class that holds the dossier properties, such as dossier number) and subfolders that contain the dossier’s documents. In a way, the top-level folder represents the whole dossier.

Fictional Company A defines the data model of a customer dossier and how the user interface should look, as in Figure 4-1.

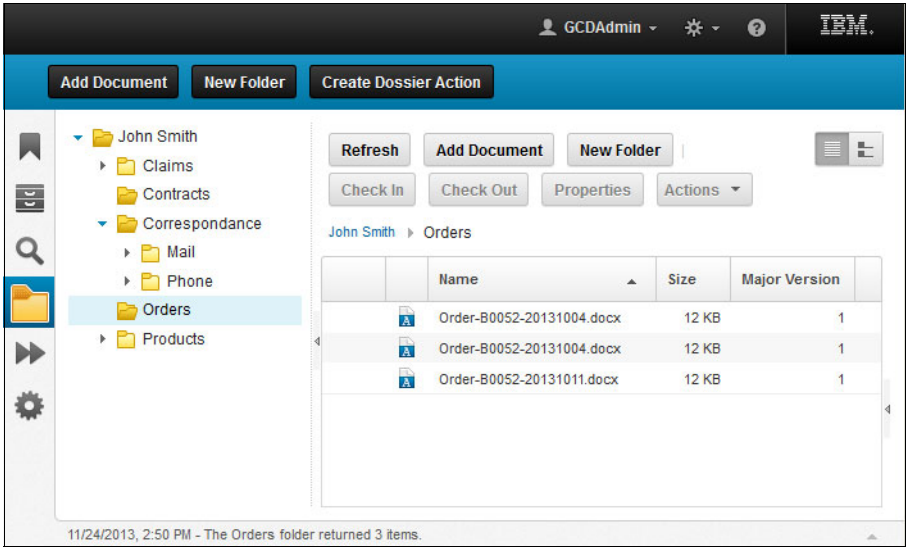



*Figure 4-1 User interface for customer dossiers*

The workers of Fictional Company mainly perform the following activities with the dossiers of their customers:

- Create a new dossier for new customers.
- Search for a customer, for example with the customer number as the search criteria, and open the customer's dossier and work on the customer documents:
  - Browse the customer documents.
  - View some of the documents.
  - Edit the metadata.
  - Add the dossier to the favorites.

We must provide the following extensions:

- An action to create a new customer dossier

  The create action includes a midtier service to create the substructure of the dossier.
- An action to open a customer's dossier and show the dossier view (see Figure 4-1 on page 118)
- A feature for the dossier view
- A configuration pane so users can define the following items:
  - The folder class for the dossier (the top-level folder for the dossier)
  - The folder in which all the dossiers are filed
  - The folder that holds the template for the folder substructure of all dossiers

In this chapter, the plug-in is developed in an iterative process (we do not implement everything and pack and deploy the solution once and at the end). A step-by-step procedure is provided and we pack and deploy the solution at various times to immediately see the progress we make. With this approach, troubleshooting is much easier if an error occurs because we can always easily return to the last iteration when all was working well; we must analyze only the last modifications we made.

We use IBM Rational Application Developer and the Eclipse Plugin Extension for IBM Content Navigator development as the integrated development environment (IDE). The menus and steps in Eclipse are similar to those that are described for IBM Rational Application Developer. Although not absolutely necessary, we suggest you use the Eclipse Plugin Extension for IBM Content Navigator development for this chapter. All the generated code can also be written manually but it will be easier for you to follow through this chapter if you use the Eclipse Plugin.

To download the source code for this chapter, see Appendix D, "Additional material" on page 535.

## 4.2 Developing the Create Dossier action

This section steps through the process of creating an IBM Content Navigator plug-in to create a dossier for a new customer.

The steps to add the Create Dossier action through a plug-in are summarized as follows:

1. Set up a new plug-in project.
2. Package and deploy.
3. Add the action.
4. Implement the action JavaScript.
5. Prepare the ECM system and deploy current version.

### 4.2.1 Setting up a new plug-in project

Chapter 3, “Setting up the development environment” on page 73 describes the process of creating a plug-in and the structure of the IBM Rational Application Developer project that we use here.

To set up a new plug-in project for the dossier example, the steps are as follows:

1. Create a new IBM Content Navigator Plug-in project with the values in Table 4-1.

*Table 4-1 Parameters for IBM Content Navigator plug-in project*

| Parameter | Value |
|---|---|
| Descriptive name | `Dossier Plugin` |
| Java package | `com.ibm.ecm.extension` |
| Class name | `DossierPlugin` |

2. Add necessary libraries for your back-end system to the project.
3. Modify `build.xml` by adding the new JAR files of step 2 and do further modifications if necessary.

We do not provide further details for these steps, because it is already explained in 3.3, “Plug-in development” on page 81.

Your project now looks similar to Figure 4-2.

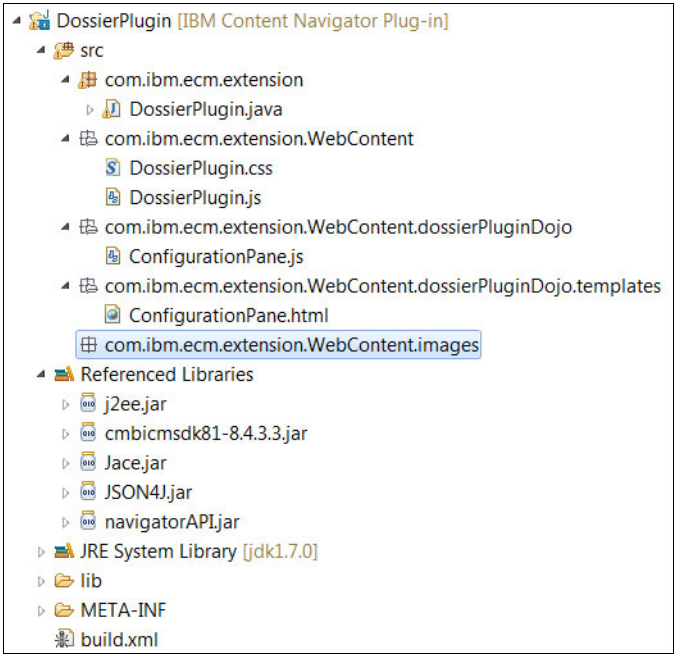



*Figure 4-2 The plug-in project including the top-level package*

The wizard created a Plugin class DossierPlugin in the `com.ibm.ecm.extension` package, which is the base class of the plug-in. In this class, we will register all the extension points we implement in this chapter.

At this point, the plug-in provides no functionality because all methods that define IBM Content Navigator's extension points have only "dummy" implementations. Throughout this chapter, we implement some of these methods to gradually add more functionality to the plug-in.

## 4.2.2 Packaging and deploying

During this iterative, step-by-step approach, we pack and deploy the plug-in now. In this way, we can confirm that we correctly set up the project and have a good starting point for further enhancements.

Instructions for how to package and deploy a plug-in to IBM Content navigator are in 3.3, "Plug-in development" on page 81. Following the steps results in an administration view of your plug-in similar to Figure 4-3 on page 122.

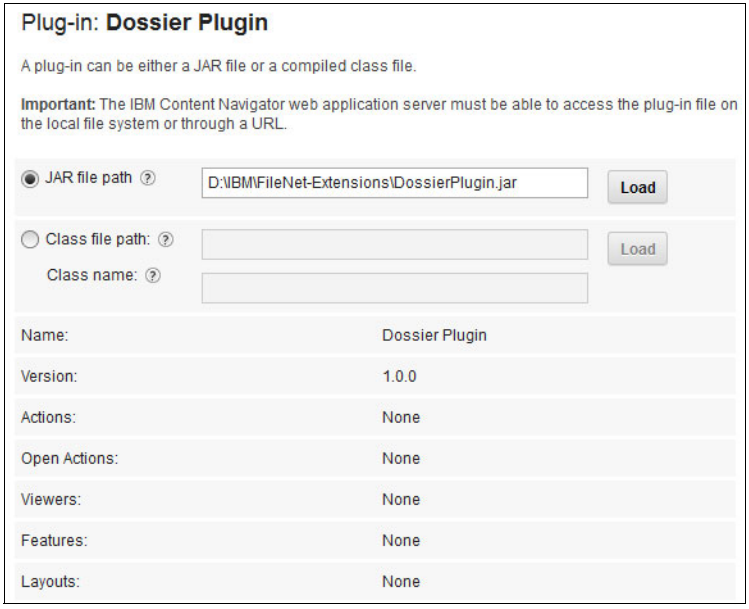



*Figure 4-3 Packing and deploying DossierPlugin: Iteration 1*

As you can see, no *extension points* of IBM Content Navigator are defined. Make sure your window is similar as the one shown in Figure 4-3.

**If your window is not similar:** if the lower part (beginning with the row of the Name plug-in) is not displayed, then your plug-in did not load. This is likely because of some errors. You must determine the cause of the problem before proceeding. One problem might be that your `Manifest.mf` file of your plug-in JAR is not defining your Plugin class correctly. This happens, for example, if you renamed your Java packages and did not update the manifest file.

See how to debug and troubleshoot IBM Content Navigator in Chapter 14, “Debugging and troubleshooting” on page 495.

## 4.2.3 Adding the action

We next implement an IBM Content Navigator action extension point. To add an action to the plug-in, complete these steps:

1. Create a specific `com.ibm.ecm.extension.PluginAction` Java class and implement its methods to add the functionality you need.
2. Hook it to the base Plugin class by adding your PluginAction class, overriding the getActions() method.

Step 1 can be done with the New Action wizard of the Eclipse Plugin for IBM Content Navigator development, as shown in 3.3.2, “Creating plug-in extension” on page 85. Use CreateDossierAction as the class name. This creates a skeleton of the class with basic implementations for each of the abstract methods. Example 4-1 demonstrates the main methods needed for the Create Dossier sample:

- getName(): Return the name of the action, which is Create Dossier Action.
- getPrivilege(): The Create Dossier action must be able to add new folders to the repository, so we specify privAddItem privilege. For a full list of available privileges, see Appendix A, “Action privileges” on page 513.
- isMultiDoc(): We are not concerned about this parameter for this action. This is because we will define the Create Dossier action to be global, which means the action can be invoked without having to select any documents or folders. The default value of `false` is acceptable.
- isGlobal(): We set the value to `true` because we want to show the Create Dossier in the global toolbar.
- getActionFunction(): This method must return the name of the JavaScript method that implements the action (which is covered in 4.2.4, “Implementing the action JavaScript” on page 124). Implementing the JavaScript function in this way will place it into global namespace which should be avoided in the “modern” Dojo world. How this can be accomplished is shown in another action we provide in 4.4, “Developing the Open Dossier action” on page 145. But as the current API still instructs to do it in this way, we also show it this way.
- getServerTypes(): We provide support for IBM FileNet P8 and IBM Content Manager, so set the value to `p8` and `cm`.
- getActionModelClass(): This method allows you to override the Action model (`ecm.model.Action`). For this action, we use the default implementation, but for the Open Dossier action, we provide a custom Action model, see 4.4.2, “Extending the Action model” on page 147.

*Example 4-1 Implementation of the CreateDossierAction class*

```
public class CreateDossierAction extends PluginAction {
   public String getName(Locale locale) {
      return "Create Dossier Action";
   ...
   public String getPrivilege() {
      return "privAddItem";
   }
   public boolean isMultiDoc() {
      return false;
   }
```

```
   public boolean isGlobal() {
      return true;
   }
   public String getActionFunction() {
      return "createDossierAction";
   }
   public String getServerTypes() {
      return "p8,cm";
   }
      public String getActionModelClass() {
      return "";
   }
}
```

Step 2 is already accomplished by the New Action wizard. See the implementation of the getActions() method in DossierPlugin class in Example 4-2.

*Example 4-2 Adding an ActionPlugin to the Plugin class.*

```
public com.ibm.ecm.extension.PluginAction[] getActions() {
   return new com.ibm.ecm.extension.PluginAction[] {
      new com.ibm.ecm.extension.CreateDossierAction()
   };
}
```

### 4.2.4 Implementing the action JavaScript

The createDossierAction JavaScript method that is provided in this section is invoked when a user performs the new action. The method that we need for our implementation include the following parameters:

- repository: Provides a link to the current repository.
- items: This is an array of `ecm/model/item` objects to which this action should be applied to. In our case, because we use the action in the global toolbar, this parameter is undefined. An alternative approach is to define the action in a folder context menu and let the user invoke the Create Dossier action only while selecting a parent folder. In that case, the items array are filled with the selected folder item.
- callback: You can provide any JavaScript function that will be called after the action method has been completed.

The task of the createDossierAction method can be divided into two subtasks:

1. Create a new folder that will be the top-level folder of the new dossier.
2. Create the substructure of the top-level folder, which is a subfolder structure, defined in a template structure.

This section shows the first task; the second task is describe in 4.3, “Developing a plug-in service to create the substructure of the dossier” on page 133.

For the creation of a new folder we can reuse the generic widget, `ecm/widget/dialog/AddContentItemDialog`, which is provided by IBM Content Navigator to add all types of items to a repository.

**Alternate approach:** Build your own dialog by extending `ecm.widget.dialog.BaseDialog`. To keep the process simple, consider which existing widget of IBM Content Navigator comes closest to your requirements and try to adapt that component to your needs. If this is not feasible, then use your own implementation.

We prepare this dialog with the following settings:

- Set the parent folder of the new dossier. This parent folder will be the parent folder of all our dossier folders. So we call it `dossierRootFolder` and set it to a hard-coded folder, which is made configurable in a later section.
  - For FileNet P8, dossierRootFolder is specified with a folder path.
  - For Content Manager, dossierRootFolder is specified with the persistent identifier (PID) of an item. One way to get the correct PID is to navigate to the item in the Browse pane of IBM Content Navigator and open its properties pane. In the System Properties area, you can find the ID value, as in the following example:

    ```
    96 3 ICM15 cm-richmondvm129 ClbFolder59 26
    A1001001A13E14B45839G3889018 A13E14B45839G388901 14 1009
    ```

**Source code for IBM FileNet P8 and IBM Content Manager:**
This chapter is implemented for both IBM FileNet P8 and IBM Content Manager. Because the code here was implemented for FileNet P8 at first, the terminology at some point in the source code might be specific to FileNet P8.

If implementation details are specific for IBM Content Manager, they are described in a comment in the source code. So, if your platform is IBM Content Manager you scan the source code for comments starting with the following text:

`//CM code: ....`

- Set the folder class to the folder class of the dossiers, which is CustomerDossier in the hard-coded version.

  For Content Manager, set the name of the item type of the dossier.
- Change the title of the dialog to `Create new Dossier`.
- Change the introduction text of the dialog to a meaningful description.

Before creating the dialog, retrieve the parent folder of the dossier. For retrieving various items, you can use `ecm.model.Repository`, which provides the method retrieveItem(). The first parameter is the path of the new folder; second parameter is the callback function that is invoked when the retrieveItem() method is returned. The retrieveItem() method retrieves the specified item by path and returns an `ecm.model.ContentItem`, which is passed as a parameter to the callback() function; see Example 4-3.

**Note:** The retrieveItem() method is executed asynchronously. This is the usual way for all methods that go to the midtier service layer. The thread that invokes this method immediately continues with the next line of code, while the retrieving process is still in progress. So no code after this method can assume that the requested item is already retrieved. All code that must rely on the retrieved item must go into the callback function.

*Example 4-3 Implementation of createDossierAction in DossierPlugin.js: Iteration 1*

```
require(["dojo/_base/declare","dojo/_base/lang",
         "ecm/widget/dialog/AddContentItemDialog"],
function(declare, lang, AddContentItemDialog) {
   lang.setObject("createDossierAction", function(repository, items) {
      var dossierRootFolder = "/CustomerDossiers"; //CM code:folder-PID
      var dossierFolderClass = "CustomerDossier"; //CM code: item type
      var templateDossierStructure = "/TemplateDossierStructure";
```

```
         //CM code: folder-PID of item for template folder structure

      var _createFolderSubStructure = function (dossierFolder) {
         console.log("Dossier folder created: " + dossierFolder.name);
      };

      repository.retrieveItem(dossierRootFolder,
function(dossierRootFolderItem) {
         var addContentItemDialog = new AddContentItemDialog();
      addContentItemDialog.setDefaultContentClass(dossierFolderClass);
         addContentItemDialog.show(repository, dossierRootFolderItem,
false, false, _createFolderSubStructure, null, false, null);
         addContentItemDialog.set("title","Create new Dossier");
         addContentItemDialog.setIntroText("This folder will be the top
level folder of your dossier.");
      });
   });
});
```

In our sample, the callback function of the retrieveItem method is anonymous and has one parameter, dossierRootFolderItem, which will be set to the retrieved item. The main purpose of this function is to open the dialog to create the new dossier; the next, callback function is declared as `_createFolderSubStructure` function. It is invoked when the dialog is closed when the OK button is clicked and the new folder is created. The first parameter, dossierFolder is filled with the newly created folder.

The creation of the substructure of the dossier depends on the newly created top-level folder of the dossier. That is why that code must be executed in the `_createFolderSubStructure` callback function. For this iteration, we print only a message to the console.

## 4.2.5 Preparing ECM system and deploying current version

Before redeploying the plug-in with the new action, you must prepare the ECM back-end system. First, prepare an IBM FileNet P8 system. Then, prepare an IBM Content Manager system. Then deploy the current plug-in version.

### Preparing your IBM FileNet P8 system

Prepare a FileNet ObjectStore of your choice. You can use either of the following ways:

- Import the export manifest file of the `exportP8.zip` archive (which is provided in Appendix D, “Additional material” on page 535) into your ObjectStore

- Manually prepare the following items:

  a. Create a P8 folder class with name CustomerDossier and add StringType properties such as CustomerNumber, Company, Phone Number, or others.

  b. Create a P8 folder directly under the root folder with the name `CustomerDossiers` (so the path to this folder is `/CustomerDossiers`). All the customer dossiers will be filed in this folder.

  c. Create a P8 folder directly under the root folder with the name `TemplateDossierStructure` (so the path to this folder is `/TemplateDossierStructure`). This is the template folder structure, which will be shared by all dossiers.

  d. Create a subfolder under `/TemplateDossierStructure` and use names such as `Contracts`, `Claims`, `Orders`, or others. You can also create level-two folders.

     This folder structure will be copied to all of your dossiers by the service we implement in 4.3, "Developing a plug-in service to create the substructure of the dossier" on page 133.

Be sure that your IBM Content Navigator users have proper access rights:

- For the root folder of the customer dossiers: At least *add to folder* access level.
- For the template folder (and subfolders): At least *view properties* access level.
- For the folder classes CustomerDossier and Folder: At least *create instance* rights.

## Preparing your IBM Content Manager system

Prepare your Content Manager repository with the following steps:

1. Create an item type with name CustomerDossier. We use this item type to create folder items that serve as a top-level folder of a dossier.

   Add attributes such as CustomerName, CustomerNumber, Company, Phone Number, or others.

2. Create a folder item with the name `CustomerDossiers`. All the customer dossiers will be children of this folder.

   If you want this item type to have been preselected when you create a new dossier, choose the **CustomerDossier** item type for Customer Dossiers. The reason is because, for Content Manager, the AddContentItemDialog preselects the item type with the one from the selected parent folder.

3. Create a folder item with the name `TemplateDossierStructure`. This folder item is the template folder structure, which will be shared by all dossiers.
4. Create any subfolder items as child folders of `TemplateDossierStructure`, such as `Contracts`, `Claims`, `Orders`, and others. You can also create level-two folder items.

   This folder structure will be copied to all of your dossiers by the service we implement in 4.3, “Developing a plug-in service to create the substructure of the dossier” on page 133.

Be sure that your IBM Content Navigator users have appropriate permissions:

- For the root item folder of the customer dossiers: At least permission to add to this folder item.
- For the template folder item (and subfolders items): At least permission to view the items and its attributes.
- Permissions for the users to create items as folders for the item types:
  - CustomerDossier (for creating top level folders of the dossier)
  - <Item type> of the folder items of the template structure (for creating the subfolder structure of the dossier).

## Deploying the plug-in with the action

At this point, we deploy our plug-in again (as in 4.2.2, “Packaging and deploying” on page 121). The Actions section of the plug-in pane now shows CreateDossierAction (Figure 4-4).

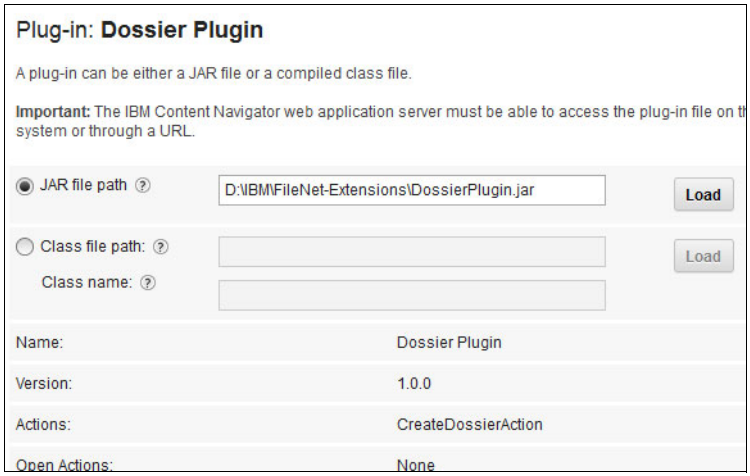



*Figure 4-4 Dossier Plugin with CreateDossierAction*

The next step is to configure the Create Dossier action so it is displayed in the global toolbar of IBM Content Navigator. To add the new action to the toolbar (or menu), complete the following steps in the Administration view of IBM Content Navigator:

1. Select **Menus** from the list on the left. A list of the menus that are defined is displayed. For our example, we select the **Default global toolbar** from the list.
2. Because the default menus are read-only, make a copy of the toolbar to modify it. To do so, open the context menu of the Default global toolbar and click **Copy**. The New Menu dialog box opens (Figure 4-5).
3. In the New Menu dialog box, enter `Dossier global toolbar` as the name of the new menu.
4. From the Available Action list on the left panel, select **CreateDossierAction** and move it to the right-side panel of Selected Actions.

   Optional: You can add a separator before the new action or remove actions that you do not want to appear in the global toolbar.

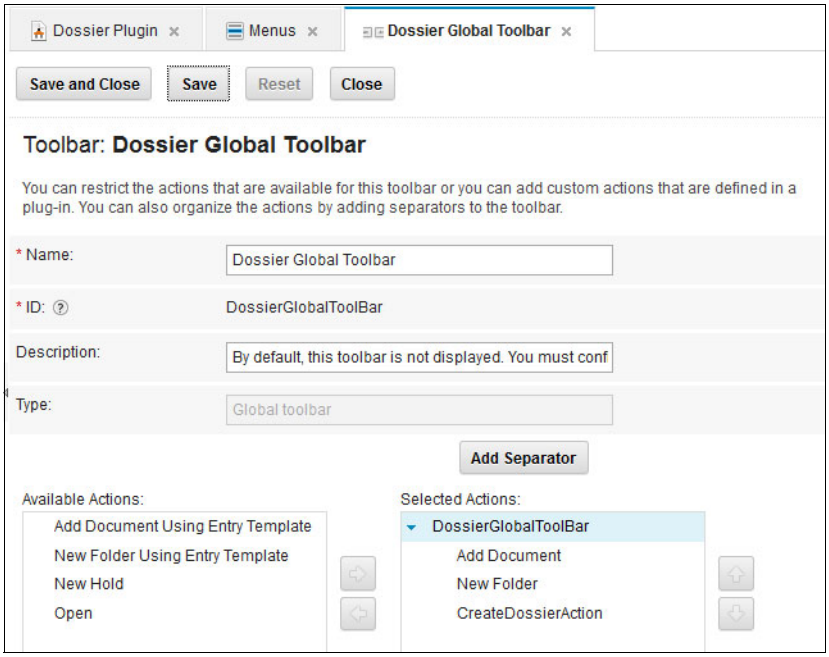



*Figure 4-5 Global Toolbar with CreateDossierAction*

After a menu is created, it must be added to a desktop so that it can be displayed. You can add the new menu to the desktop of your choice. We create a new desktop with the name DossierDesktop with following specifications:

- In the Administration view, select **Desktops** → **New Desktop**.
- From the General tab, enter the following information:
  - Name and ID: DossierDesktop
  - Authentication: Select the ObjectStore you prepared in the previous session.
- From the Repositories tab, assign a repository that we prepared in previous subsection.
- From the Appearance tab; use these settings:
  - Application name: Simple Dossier Management.
  - Displayed features: Select Favorites, Browse, Search.
  - Default repository: Select the repository you just assigned to the desktop.
  - Show global toolbar (at the bottom): Enable.

To add the global toolbar to the desktop, complete the following steps:

1. Select the **Menus** tab to assign a new menu.
2. Find the global toolbar and in the associated drop-down list, select **Dossier Global Toolbar.** Because our toolbar is a copy of the default content list toolbar, it is displayed in the drop-down list. See Figure 4-6.
3. Click **Save** to submit the changes to the desktop configuration.

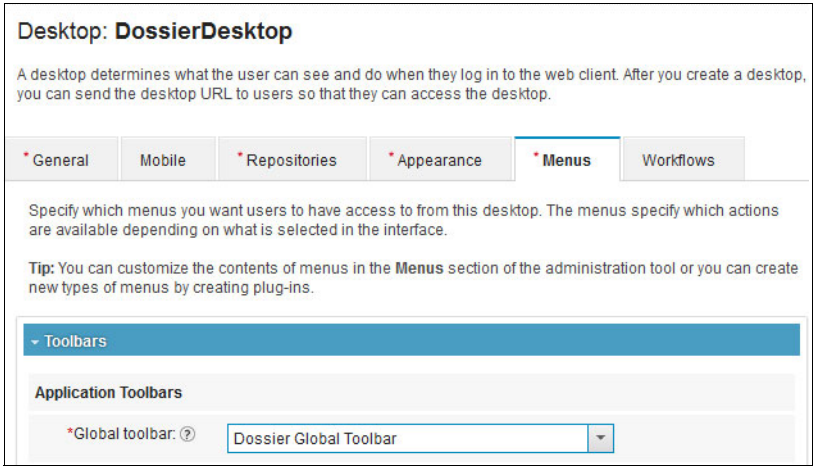



*Figure 4-6 Set Dossier Global Toolbar as the global toolbar*

Now we are ready to see our new action in the global toolbar:

1. Empty the cache from your browser (use the shortcut: Ctrl+Shift+Del for Firefox and Internet Explorer).
2. Enter a URL such as the following example:

   `http://<webserver>:<port>/navigator/?desktop=DossierDesktop`
3. Log in as a user with appropriate permissions. You will see the Create Dossier action in the global toolbar.

**Troubleshooting:** If you do not see the global toolbar, be you selected the check box in the configuration of the desktop at the bottom of the Appearance tab.

If you see the global toolbar but not the Create Dossier action, verify that you assigned the global toolbar with the Create Dossier action selected to the desktop to which you logged in.

Click the **Create Dossier** button to open the Create Dossier dialog. See Figure 4-7 on page 133. The folder class and the parent folder are preselected. Remember, for Content Manager repositories, the preselected class is not the specified item type defined in “Preparing your IBM Content Manager system” on page 128, but the item type of the parent folder item.

Provide meaningful values for the properties of the dossier and click **OK**. A new folder is created as a child of the parent folder (`/CustomerDossiers`). To verify the result, switch to the browse feature and navigate to your new folder.

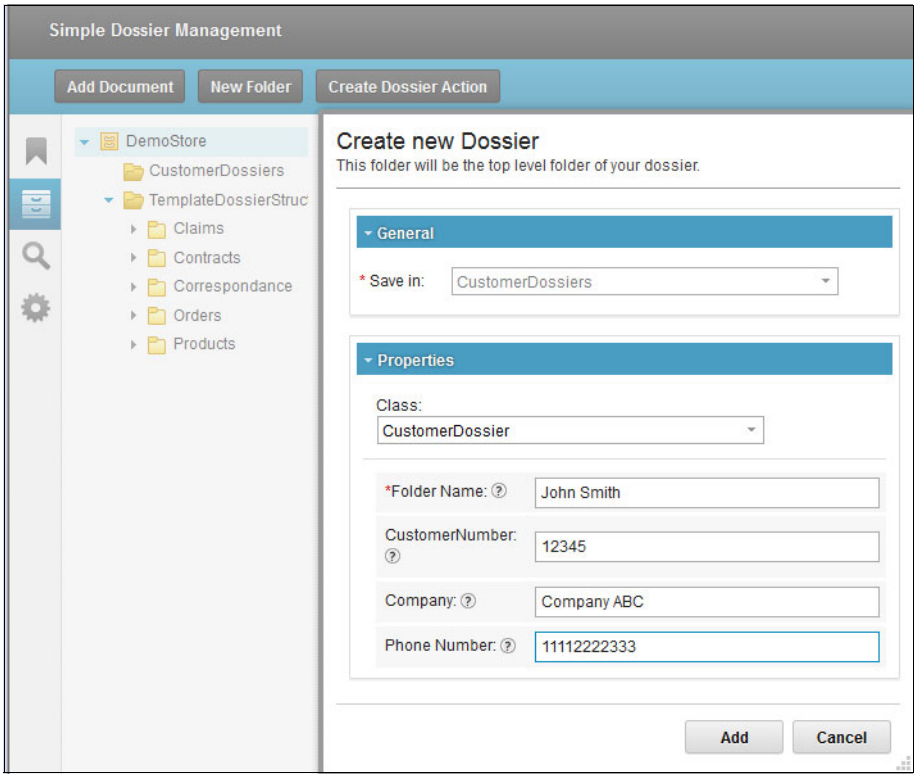



*Figure 4-7 Create new Dossier dialog*

## 4.3 Developing a plug-in service to create the substructure of the dossier

If the plug-in must provide back-end functionality, a service must be implemented. Often, the service is integrated to an action because the user cannot directly invoke a service in the user interface of IBM Content Navigator; therefore, the user must call an action that will invoke a service.

This example shows how you can develop a service to create the substructure of the dossier.

To add a service to the plug-in, do these steps:

1. Create a specific com.ibm.ecm.extension.PluginService Java class and implement its methods to add the functionality you need.

2. Hook it to the general Plugin class by adding your PluginService class overriding the getServices() method.

Both tasks can be done with the Eclipse Plugin for IBM Content Navigator development, as shown in 3.3.2, “Creating plug-in extension” on page 85. Use CreateSubStructureService as class name. This creates the class with skeleton and basic implementations for each of the abstract methods. See Example 4-4.

*Example 4-4 Skeleton of the PluginService*

```
public class CreateSubStructureService extends PluginService {
   public String getId() {
      return "CreateSubStructureService";
   }
   public String getOverriddenService() {
      return null;
   }
   public void execute(PluginServiceCallbacks callbacks,
HttpServletRequest request, HttpServletResponse response) throws
Exception {
   }
}
```

Example 3-6 shows how the wizard has registered the new service into the DossierPlugin class.

*Example 4-5 getService() of DossierPlugin class*

```
public com.ibm.ecm.extension.PluginService[] getServices() {
      return new com.ibm.ecm.extension.PluginService[]{
         new com.ibm.ecm.extension.CreateSubStructureService()
      };
}
```

Before we implement the server type specific code, we integrate our CreateSubStructureService service into the Create Dossier action. In the Create Dossier action in DossierPlugin.js, we already have prepared the `_createFolderSubStructure` function. Example 4-6 on page 135 shows the new implementation of it.

We prepare the parameters for the CreateSubStructureService service in the serviceParams object and use the invokePluginService function, which is provided by IBM Content Navigator to use for any midtier service that is provided by a plug-in. The first parameter specifies the plug-in; the second parameter specifies the ID of the service.

Do not forget to define the additional Dojo module we need: `ecm.model.Request`.

*Example 4-6 Implementation of createDossierAction in DossierPlugin.js*

```
require(["dojo/_base/declare","dojo/_base/lang",
         "ecm/widget/dialog/AddContentItemDialog",
         "ecm/model/Request"],
function(declare, lang, AddContentItemDialog, Request) {
...

var _createFolderSubStructure = function (dossierFolder) {
   var serviceParams = {
      icnRepository : repository.id,
      serverType : repository.type,
      dossierId : dossierFolder.id,
      templateFolderStructure: templateDossierStructure
   };
   Request.invokePluginService("DossierPlugin",
"CreateSubStructureService",
      {
         requestParams: serviceParams
      }
   );
};
```

At this point, the service is invoked after the top-level folder of the dossier is created. Now that all is prepared, we can implement the actual service.

The main method we must implement for the service is `execute`. In our example, the service creates a folder structure beneath the top-level folder of the new dossier.

The implementation of the service consists of two parts:

- A part that independent of server type: extracting the parameters, preparing and invoking the service call specific to the back end
- A part that depends on the selected server type: the actual implementation of the back-end service

Therefore, we separate the code into different files: one is server-independent and then there is one implementation file for each supported server type. We describe each file in the next subsections. The server-type-specific code is not explained in detail because the focus of this book is IBM Content Navigator development and not back-end implementation. We also assume that if you need to develop code to work with a specific server type, you must be familiar with development with that server type.

### 4.3.1 Server type independent code: CreateSubStructureService

The execute() method starts by extracting the parameters that are passed in. See Example 4-7. Do not forget to add necessary Java import statements (as a shortcut, you can use Ctrl+Shift+O)

*Example 4-7 Implementation of CreateSubStructureService*

```
public void execute(PluginServiceCallbacks callbacks,
HttpServletRequest request, HttpServletResponse response) throws
Exception {
      callbacks.getLogger().logEntry(this, "execute", request);
      String icnRepositoryId = request.getParameter("icnRepository");
      String serverType = request.getParameter("serverType");
      String dossierId = request.getParameter("dossierId");
      String templateFolderStructurePath =
request.getParameter("templateFolderStructure");

      JSONResponse jsonResults = new JSONResponse();
      try {
         if ("p8".equals(serverType)) {
            CreateSubStructureServiceP8.execute(callbacks, jsonResults,
dossierId, templateFolderStructurePath, icnRepositoryId);
         } else if (serverType.equals("cm")) {
            Object synchObject =
callbacks.getSynchObject(icnRepositoryId, serverType);
            if (synchObject != null) {
               synchronized (synchObject) {
                  CreateSubStructureServiceCM.execute(callbacks,
jsonResults, dossierId, templateFolderStructurePath, icnRepositoryId);
               }
            } else {
               CreateSubStructureServiceCM.execute(callbacks,
jsonResults, dossierId, templateFolderStructurePath, icnRepositoryId);
            }
         } else {
            JSONMessage jsonMessage = new JSONMessage(0, "Server type
is not supported: "+serverType,"", "", "Not yet implemented.", "");
            jsonResults.addErrorMessage(jsonMessage);
         }
      } catch (Exception e) {
         callbacks.getLogger().logError(this, "execute", request, e);
         JSONMessage jsonMessage = new JSONMessage(0, "Creation of the
dossier's substructure failed.", e.getMessage(),"","Check the IBM
Content Navigator logs for more details.", "");
```

```
            jsonResults.addErrorMessage(jsonMessage);
        }
        jsonResults.serialize(response.getOutputStream());
        callbacks.getLogger().logExit(this, "execute", request);
    }
```

When you implement the action and the service, you must determine which parameters are needed to implement the functionality. For our example, the following parameters are passed to the service in the request object:

- icnRepositoryId: Identifies the repository in which the new dossier will be stored.
- serverType: Identifies the target back-end server type. For our service type, valid values are `p8` and `cm`.
- dossierId: This is the item ID of the newly created top-level folder.
- templateFolderStructurePath: Specifies a folder in the back-end repository with a subfolder structure that will serve as a template for the dossier. Beneath the top-level folder of each dossier, these folder structure is created, so at the end, all dossiers will have the same substructure.

**Note:** The dossierId parameter is an IBM Content Navigator identifier (ID); do not confuse it with the identifiers of the underlaying ECM server type. It has a different syntax for each server type, so it is handled in the server type specific code.

- For IBM FileNet P8, the syntax is as follows:

  *<classId>*,*<objectStoreId>*,*<objectId>*
- For IBM Content Manager, the syntax is the same as the persistent identifier (PID).

After retrieving the parameters, the appropriate service implementation, depending on the server type, is called. The back-end-specific code is handled in `CreateSubStructureServiceP8.java` and `CreateSubStructureServiceCM.java`. If you want to support only one of these server types, comment the invocation of the unsupported server type in your code. Otherwise an error occurs because the code looks for both implementation classes.

The service returns a JSON response. IBM Content Navigator provides the `com.ibm.ecm.json` Java package with some helper classes, for your convenience.

In our example, we use JSONResponse, which can be used to provide three kinds of messages to the front end:

- addInfoMessage: This message is displayed in the status bar of IBM Content Navigator.
- addWarningMessage: This warning is displayed as a pop-up window.
- addErrorMessag: This error message is displayed as a pop-up window.

If the service must return some data to the front end, any property can be added to the response object. Because our service creates only the substructure and does not return anything, we add only an information message.

Finally, the serialize method of JSONRespon**se** converts the object into a stream of JSON text, which is written to the output stream of the HTTP response object.

### 4.3.2 IBM FileNet P8 code: CreateSubStructureServiceP8

The server type specific code for IBM FileNet P8 is handled in the Java class `CreateSubStructureServiceP8.java`. So, in your development environment, create a new Java class in the `com.ibm.ecm.extension` package and give it the following name: `CreateSubStructureServiceP8`.

The class has a static method with the name `execute`, which is the main entry point and performs the following steps, as shown in Example 4-8 on page 139:

1. Authenticates to IBM FileNet Content Platform Engine.
2. Fetches the ObjectStore.
3. Parses the IBM Content Navigator identifier.
4. Fetches the newly created folder (dossier top-level folder) from the ObjectStore. This will be the parent folder for the substructure.
5. Fetches the template folder which will hold the folder structure to be created.
6. Invokes the work performer createDossierSubstructureP8, which recursively creates the subfolder structure.
7. Returns the response.

To make the example more readable, only limited error and exception handling exists in this code. For production quality code, you must add more error handling. The sample code assumes, for example, that templateFolderStructurePath is a valid path name to a folder that exists in the configured ObjectStore; no error checking is done on the folder path.

*Example 4-8 Implementation of* CreateSubStructureServiceP8 with execute

```
package com.ibm.ecm.extension;
import java.util.Iterator;
import java.util.StringTokenizer;

import javax.security.auth.Subject;
import com.filenet.api.collection.FolderSet;
import com.filenet.api.constants.RefreshMode;
import com.filenet.api.core.Factory;
import com.filenet.api.core.Folder;
import com.filenet.api.core.ObjectStore;
import com.filenet.api.util.UserContext;
import com.ibm.ecm.json.JSONMessage;
import com.ibm.ecm.json.JSONResponse;

public class CreateSubStructureServiceP8 {
   public static void execute(PluginServiceCallbacks callbacks,
JSONResponse jsonResults, String dossierId,
      String templateFolderStructurePath, String icnRepositoryId) {
      Subject subject = callbacks.getP8Subject(icnRepositoryId);
      UserContext.get().pushSubject(subject);
      try {
         ObjectStore objectStore =
callbacks.getP8ObjectStore(icnRepositoryId);
   StringTokenizer dossierIdTok = new StringTokenizer(dossierId, ",");
         String classID = (dossierIdTok.hasMoreTokens() ?
dossierIdTok.nextToken() : null);
         String objectStoreID = (dossierIdTok.hasMoreTokens() ?
dossierIdTok.nextToken() : null);
         String dossierObjectId = (dossierIdTok.hasMoreTokens() ?
dossierIdTok.nextToken() : null);
         Folder dossierFolder =
Factory.Folder.fetchInstance(objectStore, dossierObjectId, null);
         Folder templateRootFolder =
Factory.Folder.fetchInstance(objectStore, templateFolderStructurePath,
null);
         createDossierSubstructureP8(dossierFolder, templateRootFolder,
objectStore);
         JSONMessage infoMessage = new JSONMessage(1000, "Successfully
created dossier substructure", "",
               "Successfully created dossier substructure for Dosser "
+ dossierFolder.get_Name(), "", "");
         jsonResults.addInfoMessage(infoMessage);
      } finally {
```

```
            UserContext.get().popSubject();
        }
    }
}
```

The createDossierSubstructureP8 method is shown in Example 4-9.

*Example 4-9 Implementation of createDossierSubstructureP8 method*

```
private static void createDossierSubstructureP8(Folder dossierFolder,
Folder templateRootFolder, ObjectStore os) {
      FolderSet subFolders = templateRootFolder.get_SubFolders();
      Iterator it = subFolders.iterator();
      while (it.hasNext()) {
         Folder templateFolder = (Folder) it.next();
         String subFolderName = templateFolder.get_FolderName();
         Folder subFolder =
dossierFolder.createSubFolder(subFolderName);
         subFolder.save(RefreshMode.REFRESH);
         // call recursively
         createDossierSubstructureP8(subFolder, templateFolder, os);
      }
   }
```

It retrieves the child folders of the `templateRootFolder` folder and creates, for each, a corresponding child folder to the `dossierFolder`, which is the top-level folder of the dossier. For each created child folder, this method is invoked recursively, so the newly created child folder becomes the parent folder for the next level of recursion.

In a more sophisticated implementation, we would not only adapt the folder name from the template, but also the folder class and maybe copy documents if they are filed in the template folder structure. One use case might be templates for office documents used for consistent correspondence to the client.

### 4.3.3 IBM Content Manager code: CreateSubStructureServiceCM

The server type specific code for IBM Content Manager is handled in the Java class `CreateSubStructureServiceCM.java`. So in your development environment, create a new Java class in the `com.ibm.ecm.extension` package and give it the following name: `CreateSubStructureServiceCM`.

The class has a static method with the name `execute`, which is the main entry point and performs the following steps, as shown in Example 4-10:

1. Gets the datastore from the callback.
2. Creates DDOs for the parent folder item and the item that holds the template structure.
3. Invokes the work performer, createDossierSubstructureCM8, which recursively creates the subfolder structure.
4. Adds an information message to the JSON response object.

To make the example more readable, only limited error and exception handling exists in this code. For production quality code, you must add more error handling. The sample code assumes, for example, that the `templateFolderStructurePath` has a valid PID of an item with semantic type Folder. No error handling is done here in the sample code.

*Example 4-10 Implementation of CreateSubStructureServiceCM with execute*

```
package com.ibm.ecm.extension;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.ecm.json.JSONMessage;
import com.ibm.ecm.json.JSONResponse;
import com.ibm.mm.sdk.common.DKAttrDefICM;
import com.ibm.mm.sdk.common.DKConstant;
import com.ibm.mm.sdk.common.DKConstantICM;
import com.ibm.mm.sdk.common.DKDDO;
import com.ibm.mm.sdk.common.DKDatastoreDefICM;
import com.ibm.mm.sdk.common.DKException;
import com.ibm.mm.sdk.common.DKFolder;
import com.ibm.mm.sdk.common.DKItemTypeDefICM;
import com.ibm.mm.sdk.common.DKRetrieveOptionsICM;
import com.ibm.mm.sdk.common.DKSequentialCollection;
import com.ibm.mm.sdk.common.DKUsageError;
import com.ibm.mm.sdk.common.dkIterator;
import com.ibm.mm.sdk.server.DKDatastoreICM;

public class CreateSubStructureServiceCM {

   public static void execute(PluginServiceCallbacks callbacks, JSONResponse
jsonResults, String dossierId,
         String templateFolderStructurePath, String icnRepositoryId) throws
DKUsageError, DKException, Exception {
      DKDatastoreICM datastore = callbacks.getCMDatastore(icnRepositoryId);
```

```
      DKDDO ddoDossier = datastore.createDDOFromPID(dossierId);
      String dossierName = getRepresentsItemAttribute(ddoDossier, datastore);
      DKDDO ddoTemplate =
datastore.createDDOFromPID(templateFolderStructurePath);

      createDossierSubstructureCM8(ddoDossier, ddoTemplate, datastore);

      JSONMessage infoMessage = new JSONMessage(1000, "Successfully created
dossier substructure", "",
            "Successfully created dossier substructure for Dosser " +
dossierName, "", "");
      jsonResults.addInfoMessage(infoMessage);
   }
   private static String getRepresentsItemAttribute(DKDDO item, DKDatastoreICM
datastore) throws Exception {
      String objectType = item.getPidObject().getObjectType();
      DKDatastoreDefICM dataStoreDef = (DKDatastoreDefICM)
datastore.datastoreDef();
      DKItemTypeDefICM itemType = (DKItemTypeDefICM)
dataStoreDef.retrieveEntity(objectType);
      DKSequentialCollection attrCol = (DKSequentialCollection)
itemType.listAllAttributes();
      dkIterator iter = attrCol.createIterator();
      while (iter.more()) {
         DKAttrDefICM attr = (DKAttrDefICM) iter.next();
         if (attr.isRepresentative()) {
            return attr.getName();
         }
      }
      // no name found
      return item.getPidObject().getPrimaryId().toString();
   }
}
```

Example 4-11 shows the implementation of the createDossierSubstructureCM8 method.

It fetches a list of subfolder DDOs of the `templateRootFolder` DDO and creates for each, a corresponding child item to the `dossierFolder DDO`, which is the top-level folder item of the dossier. For each created child folder item, this method is invoked recursively, so the newly created child folder item becomes the parent folder item for the next level of recursion.

*Example 4-11 Implementation of createDossierSubstructureCM8 method*

```
   private static void createDossierSubstructureCM8(DKDDO dossierFolder, DKDDO
templateRootFolder, DKDatastoreICM datastore)
         throws Exception {
```

```
      List<DKDDO> subFolders = getSubfolder(templateRootFolder, datastore);
      System.out.println("got number subFolders: " + subFolders.size());
      Iterator<DKDDO> it = subFolders.iterator();
      while (it.hasNext()) {
         DKDDO templateFolder = it.next();
         DKRetrieveOptionsICM retrieveOptions =
DKRetrieveOptionsICM.createInstance(datastore);
         retrieveOptions.baseAttributes(true);
         templateFolder.retrieve(retrieveOptions.dkNVPair());
         String attributeName = getRepresentsItemAttribute(templateFolder,
datastore);
         System.out.println("got attributeName: " + attributeName);
         String subFolderName = (String) getAttributeValue(attributeName,
templateFolder);
         System.out.println("create Subfolder ... : " + subFolderName);
         String itemType = templateFolder.getObjectType();

         DKDDO subFolder = createChildFolder(datastore, dossierFolder,
subFolderName, attributeName, itemType);

         // call recursively
         createDossierSubstructureCM8(subFolder, templateFolder, datastore);
      }
   }
   private static List<DKDDO> getSubfolder(DKDDO folder, DKDatastoreICM
datastore) throws Exception {
      DKRetrieveOptionsICM dkRetrieveOptions =
DKRetrieveOptionsICM.createInstance(datastore);
      dkRetrieveOptions.baseAttributes(true);
      dkRetrieveOptions.linksOutbound(true);
      dkRetrieveOptions.linksTypeFilter(
         DKConstantICM.DK_ICM_LINKTYPENAME_DKFOLDER);
      folder.retrieve(dkRetrieveOptions.dkNVPair());
      short dataid = folder.dataId(DKConstant.DK_CM_NAMESPACE_ATTR,
DKConstant.DK_CM_DKFOLDER);
      if (dataid == 0) {
         throw new Exception("No DKFolder Attribute Found!  DDO is either not
a Folder or Folder Contents have not been explicitly retrieved.");
      }
      DKFolder dkFolder = (DKFolder) folder.getData(dataid);

      dkIterator iter = dkFolder.createIterator();
      ArrayList<DKDDO> subFolders = new ArrayList<DKDDO>();
      while (iter.more()) {
         DKDDO ddo = (DKDDO) iter.next();
         subFolders.add(ddo);
      }
      return subFolders;
   }
```

```
   private static DKDDO createChildFolder(DKDatastoreICM datastore, DKDDO
parentFolder, String subFolderName, String attributeName,
         String itemType) throws Exception {
      DKDDO newDDOFolder = datastore.createDDO(itemType,
DKConstant.DK_CM_FOLDER);
      short dataId = newDDOFolder.dataId(DKConstant.DK_CM_NAMESPACE_ATTR,
attributeName);
      newDDOFolder.setData(dataId, subFolderName);
      newDDOFolder.addProperty(DKConstantICM.DK_ICM_PROPERTY_PARENT_FOLDER,
parentFolder);
      newDDOFolder.add();
      System.out.println("subFolder added:" + subFolderName);
      return newDDOFolder;
   }

   private static Object getAttributeValue(String attributeName, DKDDO item) {
      Object value = null;
      try {
         // user defined attributes
         short dataId = item.dataId(DKConstant.DK_CM_NAMESPACE_ATTR,
attributeName);
         if (dataId == 0) {
            // system attributes
            dataId = item.propertyId(attributeName);
            if (dataId == 0) {
               System.out.println("Cannot find the attribute: " +
attributeName + "for object " + item.getPidObject().getIdString());
               return "no value found";
            } else {
               value = item.getProperty(dataId);
            }
         } else {
            value = item.getData(dataId);
         }
      } catch (DKUsageError e) {
         System.out.println("Cannot find the attribute: " + attributeName +
"for object " + item.getPidObject().getIdString() + " error: "
               + e.getMessage());
         return "no value found";
      }
      return value;
   }
```

In a more sophisticated implementation, we adapt the name attribute from the template folder item and also the item type and maybe copy documents if they are filed in the template folder structure. One use case might be templates for office documents that are used for consistent correspondence to the client.

### 4.3.4 Deploying and verifying

Package your plug-in and deploy it again (as described in 4.2.2, “Packaging and deploying” on page 121). Unfortunately, in the administrative pane of your plug-in, the new service extension is not displayed. So, you must try the new functionality:

1. Empty browser caches and reload the page.
2. Log in to the DossierDesktop with an appropriate user.
3. Verify the Create Dossier action.

When you click **Create Dossier** in the global toolbar, the creation of the top-level folder is now followed by the creation of the substructure of the dossier, which is implemented in this section. Watch the status bar of IBM Content Navigator at the bottom. You see the message that the substructure is created.

**Troubleshooting:** If you receive an error while executing an action in IBM Content Navigator, determine whether a midtier service is invoked (in our sample, adding a new folder ends in a corresponding midtier service). If yes, check the application server’s system out log for any error information.

For WebSphere, this is as follows:

`<Path to the WebSphere Profile>/logs/<serverName>/SystemOut.log`

To verify the result, we still must switch to the browse feature in IBM Content Navigator and navigate to the substructure of the new dossier. In the next section, we develop an action that opens the dossier directly in the browse feature.

## 4.4 Developing the Open Dossier action

One main requirement of the dossier management in our sample scenario is to provide a comprehensive view to one specific dossier. The simplest way is to use the browse feature pane included with IBM Content Navigator, and set its root folder to the top-level folder of the dossier that should be displayed.

In this section, we develop an action that opens a selected dossier in the browse pane. If executed in a different feature, for example in the result list of the search feature, the action includes switching to the browse feature.

The steps to add the Open Dossier action through a plug-in are summarized as follows:

1. Add an action to the plug-in.
2. Extend the action model.
3. Pack, deploy, and configure it.
4. Enhance the Create Dossier action to also open the dossier.

## 4.4.1 Adding an action to the plug-in

Similar to 4.2.3, “Adding the action” on page 122, we use Eclipse Plugin for Content Navigator Development for creating a new action for our dossier plug-in. We name it OpenDossierAction and implement its methods, as shown in Example 4-12:

- getId and getName: Set by the wizard.
- getPrivilege: We do not specify any privileges for the action, because when the user has the permission to see the top-level folder of a dossier, that user is also allowed to open the dossier.
- isMultiDoc: Set to `false` because the user can open only one dossier at a time.
- isGlobal: Set to `false` because a top-level folder of a dossier must be selected to perform the action. We will make this action available for the context menu for folders.
- getActionFunction: Set to empty string, because we will provide the implementation of the action in the performAction method of the action model class.
- getActionModelClass: Specifies a Dojo module that defines the action model class for the Open Dossier action.

  We set it to `dossierPluginDojo/OpenDossierAction`, which says IBM Content Navigator is to load `OpenDossierAction.js` in a directory or package `dossierPluginDojo`. We develop the OpenDossierAction model in 4.4.2, “Extending the Action model” on page 147.

*Example 4-12 Implementation of the open action PluginAction*

```
public class OpenDossierAction extends PluginAction {
   public String getId() {
      return "OpenDossierAction";
   }
   public String getName(Locale locale) {
      return "Open Dossier";
   }
```

```
   public String getPrivilege() {
      return "";
   }
   public boolean isMultiDoc() {
      return false;
   }
   public boolean isGlobal() {
      return false;
   }
   public String getActionFunction() {
      return "";
   }
   public String getServerTypes() {
      return "p8,cm";
   }
   public String getActionModelClass() {
      return "dossierPluginDojo/OpenDossierAction";
   }
}
```

The wizard also adds the OpenDossierAction class to the defined PluginActions in the DossierPlugin's getActions method. See Example 4-13.

*Example 4-13 getActions method of DossierPlugin.java*

```
public com.ibm.ecm.extension.PluginAction[] getActions() {
      return new com.ibm.ecm.extension.PluginAction[]{
            new com.ibm.ecm.extension.CreateDossierAction(),
            new com.ibm.ecm.extension.OpenDossierAction()};
}
```

## 4.4.2 Extending the Action model

Each action that is performed in IBM Content Navigator has a corresponding model, which is `ecm/model/Action`. The model defines the behavior of the action. The main methods of the action model are as follows:

- isEnabled: Determines when the button (or menu) is active. There is a default implementation, which for example checks if the type of the selected item or items is supported by this action.
- isVisible: Determines if the button (or menu) is visible in the toolbar (or in the context menu). There is a default implementation, which for example verifies that the user has sufficient privileges to perform the action and that the current repository supports that action.

- performAction: The default implementation tries to find an action handler for the action; this code is finally executed if the user performs that action.

Most default actions use a default implementation for the first two methods and invoke a corresponding method in the following class for performing the actual action:

`ecm/widget/layout/CommonActionsHandler`

For example, the checkIn action has an actionCheckIn method in CommonActionsHandler, which implements the check-in of a document.

In this section, we implement the Open Dossier action directly in the performAction method.

**Declaring action in the global space:** In the Create Dossier action (from 4.3, “Developing a plug-in service to create the substructure of the dossier” on page 133), we defined the action as a global JavaScript object in the `DossierPlugin.js` file:

```
lang.setObject(“createDossierAction”, function(repository, ...)
{...});
```

This function is then registered to the action handlers known to IBM Content Navigator. When the action is executed by the user, the default implementation of performAction can find this action and invoke it.

We do not want to “pollute” the global space because that is not a good practice. So we provide a more object-orientated way of defining the Open Dossier action: We define it in the class, to which it belongs and that is a specialized Action model class. This approach follows the principle of encapsulation, which is one of the key concepts of object-orientated programming.

In the `com.ibm.ecm.extension.WebContent.dossierPluginDojo` package, we create a new `OpenDossierAction.js` file and declare our specialized *Action* model class. See Example 4-14 on page 149, which shows the skeleton:

- `define`: Registers the new module to the AMD loader and specifies the dependent modules (`dojo/_base/declare`, `ecm/model/Action`), which we need for our implementation.
- `declare`: Specifies the Action model class, which is identified as `dossierPluginDojo.OpenDossierAction`.
- Default implementation for the three methods described above. Now, they invoke only the methods of their super class.

*Example 4-14 Implementation of OpenDossierAction.js*

```
define(["dojo/_base/declare", "ecm/model/Action"],
   function(declare, Action) {
   return declare("dossierPluginDojo.OpenDossierAction", [ Action ], {
      isEnabled: function(repository, listType, items, teamspace,
resultSet) {
         return this.inherited(arguments);
      },
      isVisible: function(repository, listType) {
         return this.inherited(arguments);
      },
      performAction: function(repository, itemList, callback,
teamspace, resultSet, parameterMap) {
         return this.inherited(arguments);
      }
   });
});
```

## Developing the isEnabled method

First, we call the default implementation of this method by using `this.inherited(arguments)`. Then, we add our own implementation.

Enable the Open Dossier action only if the selected item is a folder and if it is a top-level folder of a dossier. And because all these dossier folders are from a specific folder class (in our example CustomerDossier), we can verify if the selected item is an object of this class. Example 4-15 shows the implementation of the isEnabled method.

*Example 4-15 Implementation of isEnabled method of OpenDossierAction.js*

```
isEnabled: function(repository, listType, items, teamspace, resultSet)
{
   var enabled = this.inherited(arguments);
   if (items && items[0].isFolder && items[0].getContentClass) {
      var sameClass =
items[0].getContentClass().name=="CustomerDossier";
      return enabled && items[0].isFolder() && sameClass;
   }
   return false;
},
```

### Developing the performAction method

This method has two tasks

- In the browse feature, set the root folder of the FolderTree to the top-level folder of the selected dossier.
- Switch to the Browse feature pane.

Example 4-16 shows the code to set the root folder, which we provide as additional function for OpenDossierAction class.

*Example 4-16 Implementation of setBrowseRootFolder in OpenDossierAction.js*

```
setBrowseRootFolder : function(newRootFolder, browseFeature) {
      browseFeature.folderTree.setFolder(newRootFolder);
      // optionally set content list to the first child.
}
```

To switch between features, we must understand the component and the mechanism that is responsible for this.

The layout of IBM Content Navigator's desktop (`ecm.model.Desktop`) determines which widgets are displayed and how they are arranged. Default base implementation is `ecm.widget.layout.MainLayout`. We are interested in two aspects of MainLayout:

- `getAvailableFeatures()`: Provides the available features for the desktop. Here we find the definition of the browse feature and its identifier that we need for switching, which is browsePane.
- launchBarContainer: Is a property that is defined through `data-dojo-attach-point` in the `MainLayout.html` template. It is of type `ecm.widget.layout.LaunchBarContainer` and is the actual component that handles initialization and selection of a specific feature that is displayed.

Now that you have the necessary background information, see Example 4-17 on page 151, which shows the code to switch to the Browse feature.

The Main method (of LaunchBarContainer) to switch to a feature is as follows.:

```
selectContentPane(button, UUID, params)
```

So, we must provide three parameters:

- `button`: We get the button with the getButtonByID method.
- `UUID`: We know this identifier from the getAvailableFeatures method.
- `params`: We must specify the repository. This is necessary for the Browse feature pane.

The getContentPaneByID method returns the initialized browse feature. And finally, we invoke the setBrowseRootFolder method.

*Example 4-17 Implementation of switching to the browsePane*

```
performAction: function(repository, itemList, callback, teamspace,
resultSet, parameterMap) {
   this.logEntry("performAction", "items=" + itemList);
   var newDossier = itemList[0];
   var layout = ecm.model.desktop.getLayout();
   var featureIdToSwitch = "browsePane";
   var button =
layout.launchBarContainer.getButtonByID(featureIdToSwitch);
   var params = {};
   params.repository=repository;

   layout.launchBarContainer.selectContentPane(button,
featureIdToSwitch, params);
   var browseFeature =
layout.launchBarContainer.getContentPaneByID(featureIdToSwitch);
   this.setBrowseRootFolder(newDossier,browseFeature);
   this.logExit("performAction");
},
```

**Logging:** IBM Content Navigator provides convenient logging capabilities with the `ecm/LoggerMixin` module. Some of the common log functions are as follows:

- Log methods for each debug level, for example `logDebug(...)`, `logInfo(...)`, and others:

  `log<DebugLevel>(`*`functionName, message, extra`*`)`

- For marking the beginning and the end of the operation:

  `logEntry/logExit(`*`functionName, message`*`)`

The `LoggerMixin` is already mixed in the base module of all models (`ecm/model/_ModelObject`) and therefore is ready for use in your custom modules (here we use it in the `OpenDossierAction` model).

An alternative way of logging while in development is to write to the console object. You have several methods for each debug level, such as `console.debug()`, `console.info()`, and more. But because this is a Firebug API call, you would not use this code in production because in some browsers especially older versions of those browsers, this code might crash the browser (because the console object is undefined).

### 4.4.3 Packing, deploying, and configuring

You can now deploy the plug-in again (as done in 4.2.2, “Packaging and deploying” on page 121). The Actions section of the plug-in pane has, in addition to the CreateDossierAction, the OpenDossierAction action.

Now, we configure the new action to show up in the folder’s context menu of IBM Content Navigator to be able to open a selected dossier folder.

To add the new action to the menu, complete the following steps in the Administration view feature of IBM Content Navigator:

1. Select **Menus** from the list on the left. A list of the menus that have been defined is displayed. For our example, we select the **Default Folder Context Menu** from the list.
2. The default menus are read-only, so make a copy of the menu to modify it. Open the context menu of the Default Folder Context Menu and click **Copy**. The New Menu dialog box opens.
3. In the New Menu dialog box, enter `Dossier Folder Context Menu` as the name of the new menu.
4. From the Available Action list in the left panel, select the **Open Dossier** action and move it to the right panel of Selected Actions. See Figure 4-8 on page 153.

   Optional: You can add a separator before the new action and move the entry to the preferred position in the list. We suggest you move the action to an upper position directly below the open action.
5. Click **Save**.

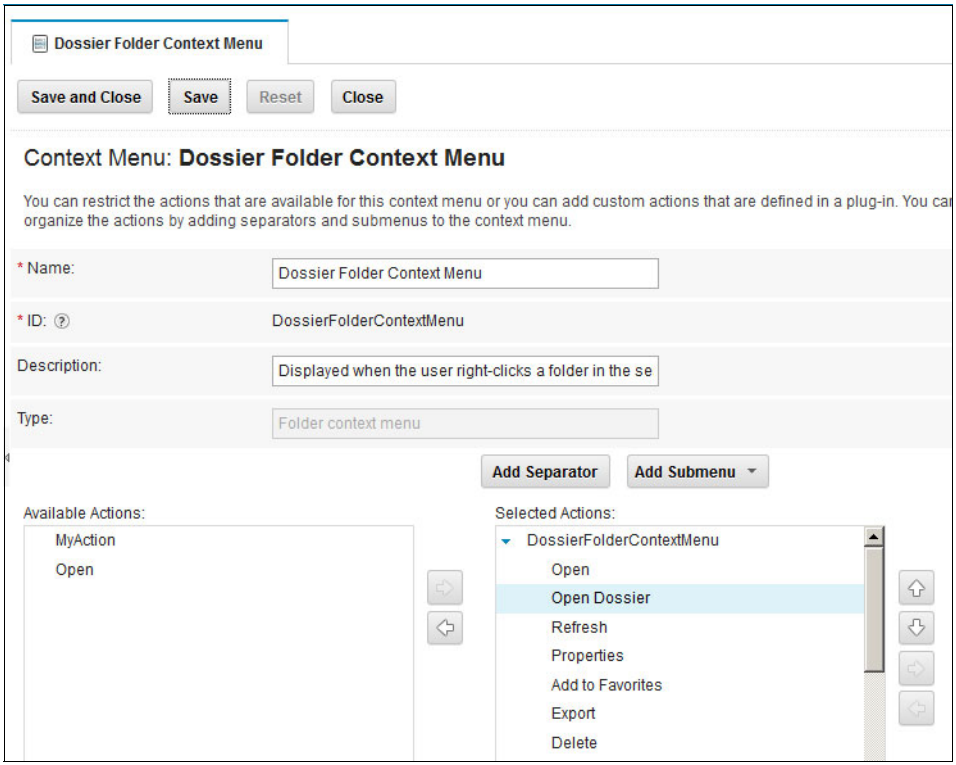



*Figure 4-8 Defining the Folder Context Menu with Open Dossier action*

**Using the dossier in the favorite feature:** If you want to use the dossiers also in the favorite feature, adding a dossier to the favorites can be done with the standard **`Add to Favorites`** action.

To open the dossier in the Favorites feature, provide your own Favorites Folder Context Menu and add the Open Dossier action to it. The steps tare similar to steps 1 - 5 on page 152 and are not shown here.

After a custom context menu is created, it must be assigned to a desktop:

1. While the Administration view is still open, select **Desktops** from the left pane. The desktops are listed in the panel on the right.
2. From the list, select the **DossierDesktop** or another desktop, and click **Edit** to display the edit window for the desktop.
3. Select the **Menus** tab to assign a new menu.

4. In the Folder context menu (Figure 4-9), select the new **Dossier Folder Context Menu** we just created. Because our menu is a copy of the Default Folder Context Menu, it is displayed in the drop-down list.
5. Click **Save** to save the changes to the desktop configuration.

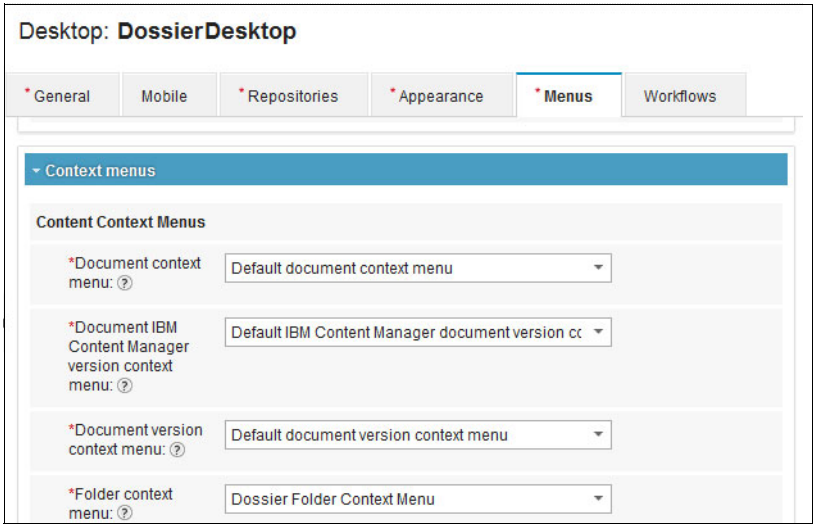



*Figure 4-9 Apply the Dossier Folder Context Menu to the desktop*

Now you are ready to see the new action in the folder context menu:

1. Empty the cache from your browser (use the shortcut Ctrl+Shift+Del for Firefox and Internet Explorer).
2. Refresh the IBM Content Navigator page (shortcut is to press F5).
3. Log in as a user with appropriate permissions.

Because you will set the root folder of the Browse feature to a dossier folder, you can no longer navigate through the whole repository. To access your dossiers, define a new search in the Search feature, which will search for dossiers. In its result list, select one dossier, right-click, and invoke the **`Open Dossier`** action. See Figure 4-10 on page 155.

Create a new search by clicking **`+New Search`** and then setting the search criteria:

- Search in Folder: `CustomerDossiers` is the root folder of all our dossiers.
- Search options: Search only folders.
- Class: Set to `CustomerDossier`.

Save the search as `Customer Dossiers`.

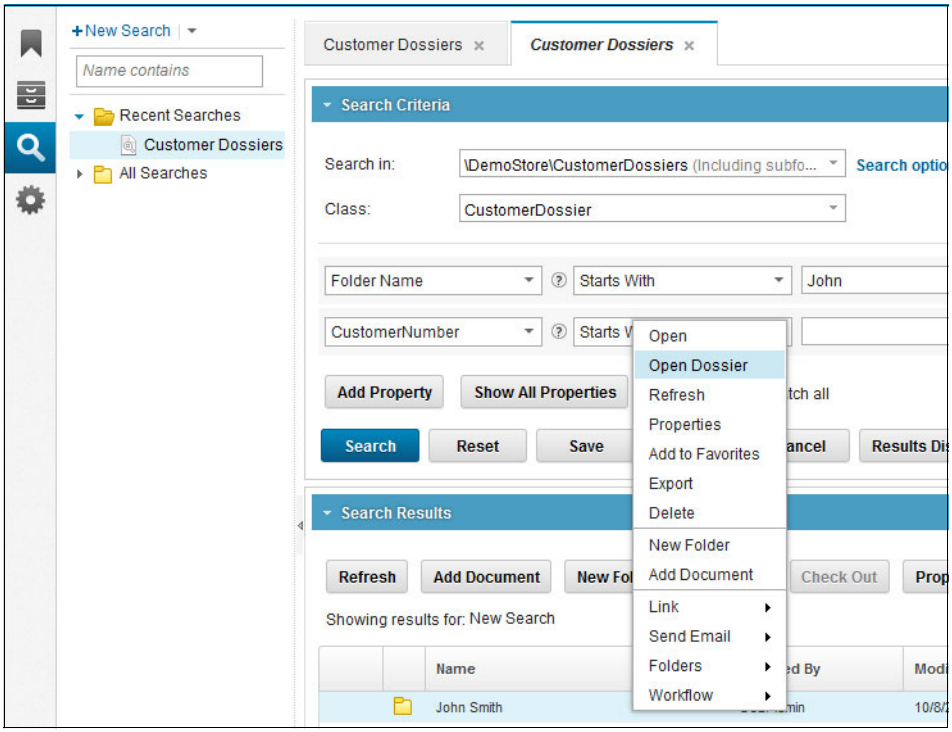



*Figure 4-10 Saved Search for Customer Dossiers*

In our example, we search for the dossier of John. From the result list, select the **Open Dossier** action; the browser feature displays, as shown in Figure 4-11. The folder tree starts with the top-level (root) folder of John Smith.

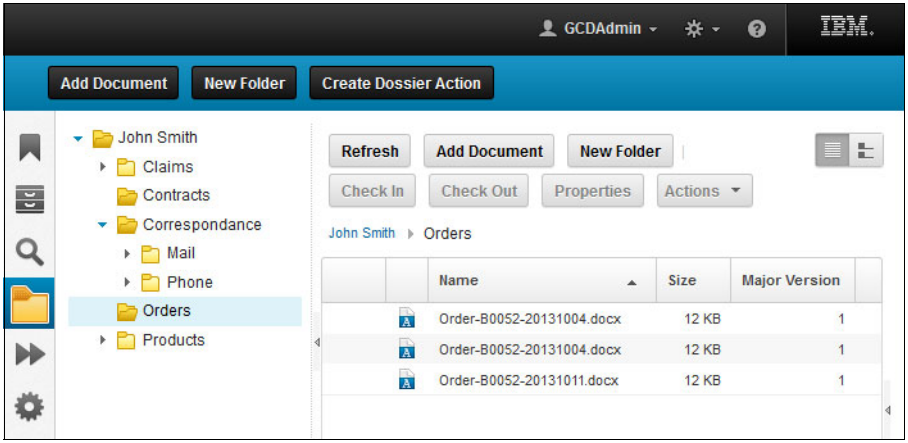



*Figure 4-11 Dossier view in the browse feature*

### 4.4.4 Enhancing the Create Dossier action to also open the dossier

With a simple enhancement, you can improve our Create Dossier action method. After the creation of the dossier, switch to the browse pane and open the newly created dossier. In the `DossierPlugin.js`, add a callback function to the invocation of our CreateSubStructureService. See Example 4-18.

*Example 4-18 enhance Create Dossier action in DossierPlugin.js.*

```
require(["dojo/_base/declare","dojo/_base/lang",
         "ecm/widget/dialog/AddContentItemDialog",
         "ecm/model/Request","dossierPluginDojo/OpenDossierAction"],
function(declare, lang, AddContentItemDialog, Request,
OpenDossierAction) {
...
Request.invokePluginService("DossierPlugin",
"CreateSubStructureService",
   {
      requestParams: serviceParams,
      requestCompleteCallback: function(response) {
         var newFolders = new Array();
         newFolders.push(dossierFolder);
         var openDossier = new OpenDossierAction();
         openDossier.performAction(repository, newFolders, null);
      }
   });
```

Do not forget to add the `dossierPluginDojo/OpenDossierAction` module to the list of required modules.

You can redeploy your plug-in now and verify that the Create Dossier action opens the new dossier directly after its creation or do it later.

This enhancement reveals the object-orientated character of the OpenDossierAction: Instantiate an action object and call its method to perform the action.

## 4.5 Open dossier view in its own feature

One unusual aspect of our solution is that we can no longer use the standard browse feature for browsing the whole repository, which might be necessary for some use cases. This section shows an easy way to specify your own feature for displaying a specific dossier.

### 4.5.1 Adding the feature

To create a new feature, we use the wizard of Eclipse Plugin for IBM Content Navigator development. Before starting the wizard, you need an icon that represents the feature. For this sample, you can use `Files32.png`, which is included with the IBM Content Navigator in the following directory:

`ECMClient\configure\explodedformat\navigator\ecm\widget\resources\images`

Copy this image to the following directory:

`src\com\ibm\ecm\extension\WebContent\images\`

Rename the file to `DossierFeature32.png`. Now start the New Feature wizard with values listed in Table 4-2.

*Table 4-2 Values to enter when creating a new feature using the wizard*

| Parameter | Value |
|---|---|
| Java Package | com.ibm.ecm.extension |
| Class Name | DossierViewFeature |
| Icon Style Class | dossierViewIcon |
| Feature Image | \src\com\ibm\ecm\extension\WebContent\images\DossierFeature32.png |

To select the Feature image, you must select **Use new Feature image**, which enables the **Select feature image** button. Click on that button, navigate to the `DossierFeature32.png` file, and select it. If no errors exist, the icon is displayed next to the Feature Image label.

The wizard generates the following files:

- `DossierViewFeature.java`: The Java class that defines the feature.
- `DossierViewFeature.js`: The Dojo module that implements the feature widget. It is referenced by the getContentClass() method in `DossierViewFeature.java`.
- `DossierViewFeature.html`: The template file that defines the layout of features with the template mechanism of Dojo. The wizard provides an empty `DIV` tag. Here is the starting point to design the layout of the feature.
- `DossierViewFeature.css`: Defines the CSS for the new feature. Currently it sets only the style for the feature's icon we provided in the wizard.

For more details about the files, see Chapter 6, "Creating a feature with search services and widgets" on page 211.

## 4.5.2 Deploying and configuring the feature

We deploy the plug-in as described in 4.2.2, “Packaging and deploying” on page 121. In the `feature` section of the plug-in administrative pane, you now find DossierViewFeature.

Next step is to assign the new feature to a desktop. Complete the following steps in the Administration view feature:

1. Select the **DossierDesktop** item from the list on the left, and open it.
2. Assign the DossierViewFeature to the desktop by moving it to the list of selected features. See Figure 4-12. If the DossierViewFeature is not in the left box, reload the page, log in, and try it again.

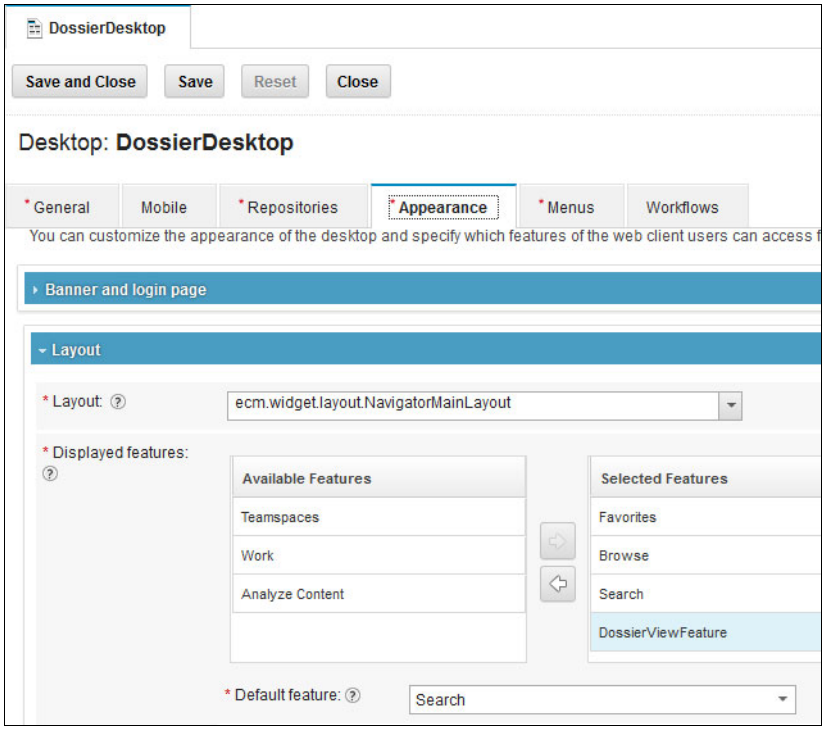



*Figure 4-12 Add the Dossier Feature to the desktop*

Log in to IBM Content Navigator, to the desktop you just configured. The icon of the new feature is in the left part. When you click the icon, an empty pane opens. See Figure 4-13.

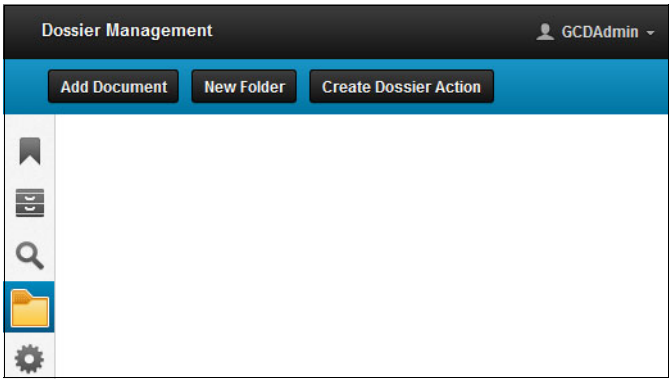



*Figure 4-13 Empty new feature*

## 4.5.3 Developing a dossier feature by using the browse feature

The New Feature wizard generates a good skeleton for creating a new feature from scratch. For example, it inherits from `_LaunchBarPane`, so it already has the functionality that LaunchBarContainer of Mainlayout can switch to the new feature.

But this time, rather than beginning from scratch, use the Browse feature and inherit all the functionality. To do so, make the `DossierViewFeature.js` empty and inherit from BrowsePane. See Example 4-19.

*Example 4-19 DossierViewFeature that inherits from BrowsePane*

```
define(["dojo/_base/declare","ecm/widget/layout/BrowsePane"],
function(declare,BrowsePane) {
   return declare("dossierPluginDojo.DossierViewFeature",
      [BrowsePane], {
   });
});
```

If you pack and deploy the plug-in now, you see the normal browse functionality when you click on the feature.

As the next step, adapt the Open Dossier action in a way that it switches to the new dossier feature and not to the default Browse feature. See Example 4-20.

*Example 4-20 Modification of OpenDossierAction.js to Open Dossier feature*

```
performAction: function(repository, itemList, callback, teamspace,
resultSet, parameterMap) {
...
var featureIdToSwitch = "DossierViewFeature";
...
}
```

At this point, you can pack and deploy the plug-in. Switch to the search pane, open the dossier search, and search for a dossier. Right-click and select **Open Dossier**. IBM Content Navigator switches to the dossier feature and opens the dossier. See Figure 4-14.

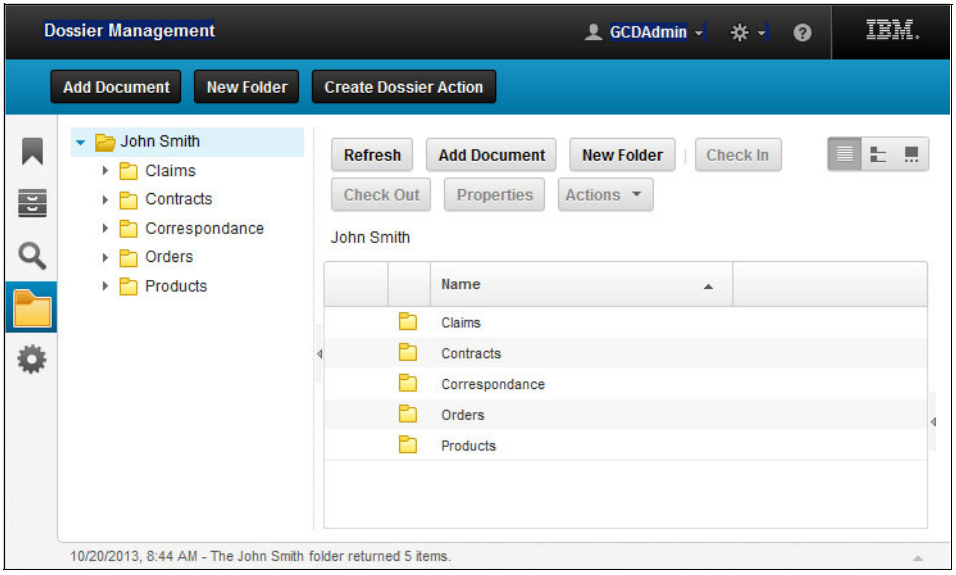



*Figure 4-14 Dossier feature with Open Dossier*

# 4.6 Adding configuration

So far, we have hard-coded several key parameters, which unnecessarily restricts the use case of the dossier we defined. In this section, we make these parameters configurable.

## 4.6.1 Adding a configuration panel

Add a configuration panel that allows you to configure the following key parameters in the sample code:

- dossierFolderClass: The name of the class of the top-level folder of the dossier. In our example, it is `CustomerDossier`. (For Content Manager, this is the name of the item type.)
- dossierRoot: The path to the parent folder of all dossiers (for example, of the top-level folder of the dossiers). In our example, it is `/CustomerDossiers`. (For Content Manager, it is the PID of the root folder item.)
- templateFolderStructure: The path to the folder that is a template for the dossier structure. Its subfolder structure will be copied to every new dossier. In our example, it is `/TemplateDossierStructure`. (For Content Manager, it is the PID of the folder item of the template structure.)

Perhaps we want to use the dossier structure for our employees. The following names might be appropriate for these parameters:

- dossierFolderClass: EmployeeDossier with properties like EmployeeNumber, Social Security number (SSN), BirthDate, EntryDate, Manager, and others.
- dossierRoot: `/EmployeeDossiers`
- templateFolderStructure: `/EmployeeDossierStructure`

IBM Content Navigator provides a mechanism for a configuration panel to be integrated with the plug-in. This panel will be displayed on the plug-in window in the Administration view and allow specific plug-in information to be specified by the administrator.

If we use the Eclipse Plugin for IBM Content Navigator development to create a new plug-in, it automatically creates a ConfigurationPane to handle the configuration for the plug-in (`ConfigurationPane.js` in `dossierPuginDojo` folder).

The ConfigurationPane inherits from the following class:

```
ecm.widget.admin.PluginConfigurationPane
```

It provides the following functionality:

- `configurationString`: A string that is managed by IBM Content Navigator. It is persisted in the database of IBM Content Navigator. You can store whatever you need for the configuration of the plug-in.
- `load`: IBM Content Navigator loads configurationString from the database. You can overload that function to parse the `configurationString`.
- `onSaveNeeded`: use this method to indicate that `configurationString` has changed and must be written to the database. If you call this method, the **Save** and the **Save and Close** buttons of the administrative pane of the plug-in are enabled.
- save: This method is automatically called by IBM Content Navigator when the Save button of the plug-in administrative pane is clicked. The `configurationString` is written to the database.

Example 4-21 shows the code for the plug-in configuration pane widget.

*Example 4-21 Implementation of ConfigurationPane.js*

```
define(["dojo/_base/declare","dijit/_TemplatedMixin",
   "dijit/_WidgetsInTemplateMixin","ecm/widget/ValidationTextBox",
   "ecm/widget/admin/PluginConfigurationPane",
   "dojo/text!./templates/ConfigurationPane.html"],
function(declare, _TemplatedMixin, _WidgetsInTemplateMixin,ValidationTextBox,
PluginConfigurationPane, template) {
   return declare("dossierPluginDojo.ConfigurationPane",
   [PluginConfigurationPane, _TemplatedMixin, _WidgetsInTemplateMixin], {
   templateString: template,
   widgetsInTemplate: true,

   load: function(callback){
      if(this.configurationString){
         var jsonConfig =  JSON.parse(this.configurationString);
this.dossierFolderClassField.set('value',jsonConfig.configuration[0].value);
this.dossierRootField.set('value',jsonConfig.configuration[1].value);
this.templateFolderStructureField.set('value',
   jsonConfig.configuration[2].value);
      }
   },

   _onParamChange: function() {
      var configArray = new Array();
      var configString = {
            name: "dossierFolderClass",
            value: this.dossierFolderClassField.get('value')
      };
      configArray.push(configString);
```

```
      configString = {
            name: "dossierRoot",
            value: this.dossierRootField.get('value')
      };
      configArray.push(configString);
      configString = {
            name: "templateFolderStructure",
            value: this.templateFolderStructureField.get('value')
      };
      configArray.push(configString);
      var configJson = {
            "configuration" : configArray
      };
      this.configurationString = JSON.stringify(configJson);
      this.onSaveNeeded(true);
   },

   validate: function() {
      if(!this.dossierFolderClassField.isValid()
         || !this.dossierRootField.isValid()
         || !this.templateFolderStructureField.isValid() )
         return false;
      return true;
   }
   });
});
```

We use the Dojo template mechanism to define the layout of the configuration pane. This is accomplished with `_TemplateMixin`, which provides templateString. The layout is defined in `./templates/ConfigurationPane.html` and is loaded with the `dojo/text!` Dojo plug-in into the templateString. Because the layout contains another widget (ValidationTextBox), you also need `_WidgetsInTemplateMixin`, which defines the widgetsInTemplate property.

The load() method parses the configurationString and grabs the values.

The next method is `_onParamChange()`. This method is called when any of the items in the configuration pane are changed. When it is called, it generates the new JSON string, based on the changed configuration values. It then calls the `onSaveNeeded(true)` method to tell the framework that the configuration values have changed. The framework then enables the Save button and also manages the save process automatically, although it can be overridden if necessary by implementing the save method.

In the validate() method, we use the validation mechanism of ValidationTextBox.

The template HTML file associated with the configuration pane defines a table with three rows. Each row contains a label and an input box, which is managed by the `ecm/widget/ValidationTextBox` widget. See Example 4-22.

*Example 4-22 HTML implementation of the configuration pane.*

```
<div class="dossierContainer" data-dojo-type="dijit.layout.ContentPane">
  <table class="propertyTable">
    <tr>
      <td class="propertyRowLabel">
       <label for="dossier_param1">Folder class of dossier</label>
      </td>
      <td class="propertyRowValue">
       <div id="dossier_param1" maxLength="128"
           data-dojo-attach-point="dossierFolderClassField"
           data-dojo-attach-event="onKeyUp: _onParamChange"
           data-dojo-type="ecm.widget.ValidationTextBox"
       data-dojo-props="required:'true',trim:'true',propercase:'false'">
       </div>
      </td>
    </tr>
    <tr>
      <td class="propertyRowLabel">
       <label for="dossier_param2">Root Folder for all Dossiers</label>
      </td>
      <td class="propertyRowValue">
       <div id="dossier_param2" maxLength="128"
           data-dojo-attach-point="dossierRootField"
           data-dojo-attach-event="onKeyUp: _onParamChange"
           data-dojo-type="ecm.widget.ValidationTextBox"
       data-dojo-props="required:'true',trim:'true',propercase:'false'">
       </div>
      </td>
    </tr>
    <tr>
      <td class="propertyRowLabel">
       <label for="dossier_param3">Template Folder for dossier substructure</label>
      </td>
      <td class="propertyRowValue">
       <div id="dossier_param3" maxLength="128"
           data-dojo-attach-point="templateFolderStructureField"
           data-dojo-attach-event="onKeyUp: _onParamChange"
           data-dojo-type="ecm.widget.ValidationTextBox"
       data-dojo-props="required:'true',trim:'true',propercase:'false'">
       </div>
      </td>
    </tr>
  </table>
</div>
```

Notice, that the Eclipse Plugin for IBM Content Navigator Development has already specified the Dojo module and the widget class for the configuration pane. Example 4-23 shows the two methods that were implemented in the `DossierPlugin.java` file.

*Example 4-23 Specification of configuration pane in DossierPlugin.java*

```
public String getDojoModule() {
  return "dossierPluginDojo";
}
public String getConfigurationDijitClass() {
  return "dossierPluginDojo.ConfigurationPane";
}
```

Figure 4-15 on page 166 shows the configuration pane of the plug-in.

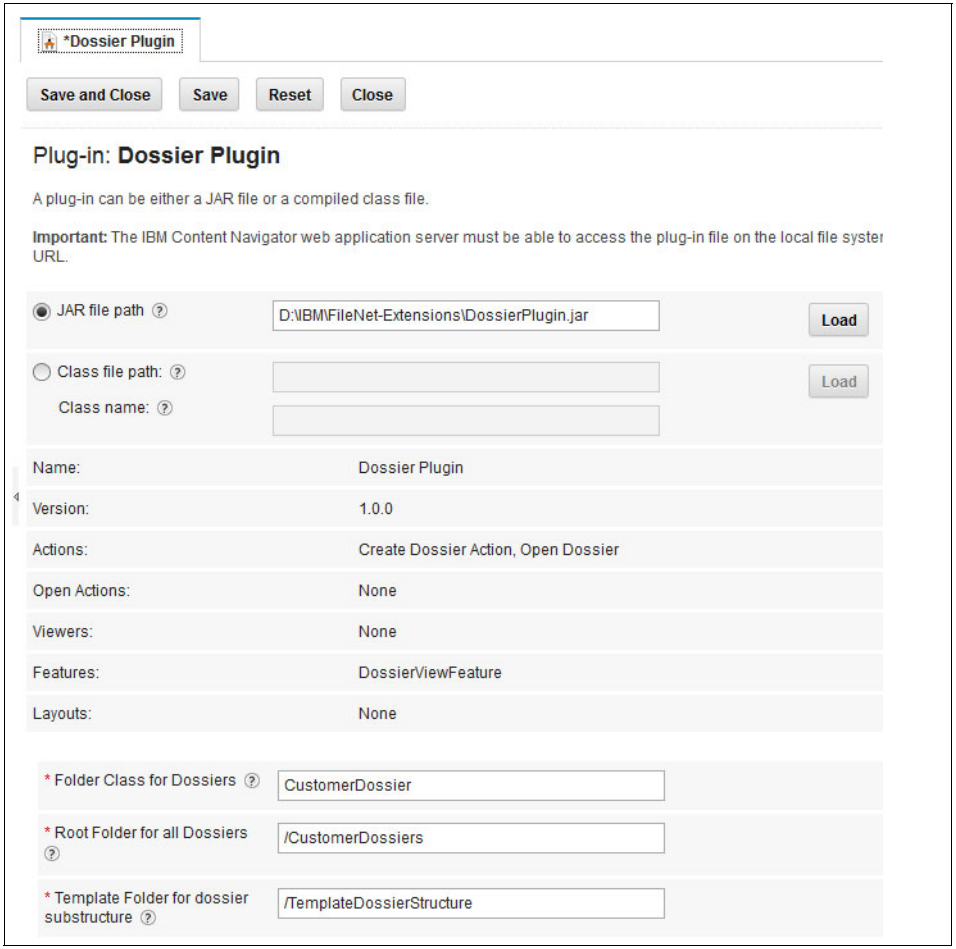



*Figure 4-15 Configuration pane for Dossier Plugin*

**Note:** In a real-life scenario, we must add a more sophisticated validation for the values the user enters in the input fields. Although the ValidationTextBox we use provides only basic validation, we can verify, for example, if the folder class or the folder paths exist in the repository. Adding extra error handling is recommended. Because only minimum validation and error handling exists now, be sure the values you provide for the three parameters are valid when testing.

Remember that the last two values for FileNet P8 are folder paths that must begin with a leading forward slash, for example `/CustomerDossiers`.

## 4.6.2 Adapting code to use configuration values

In 4.6.1, “Adding a configuration panel” on page 161, the administrator configures the three key parameters of the dossier sample. In this section, we show how the code must be adapted to remove the hard-coded parameters and use these configuration values.

The service code is independent of the three key parameter, so it remains unchanged. If it is necessary to access the configuration in the service, `com.ibm.ecm.extension.PluginServiceCallbacks` provides a method, loadConfiguration, that returns configurationString.

We must access the configuration in our two actions. There is no mechanism provided by IBM Content Navigator to directly access the configuration string of a plug-in in the JavaScript modeling library. So, we must provide a small service to fetch our configuration. We use Eclipse plugin for IBM Content Navigator development and create a new Service with the name `GetConfigurationService`.

Example 4-24 shows the implementation of the execute method. It gets the configurationString from the callback and writes it to the response object. In addition, you must import `java.io.PrintWriter`.

*Example 4-24 Implementing the GetConfigurationService*

```
public void execute(PluginServiceCallbacks callbacks,
HttpServletRequest request, HttpServletResponse response) throws
Exception {
      String configuration = callbacks.loadConfiguration();
      PrintWriter responseWriter = response.getWriter();
      try {
         responseWriter.print(configuration);
         responseWriter.flush();
      } finally {
         responseWriter.close();
      }
}
```

Now we are prepared to update the Create Dossier action.

Example 4-25 on page 168 shows the updated code for the createDossierAction() method in the `DossierPlugin.js` file. Basically, the entire code is surrounded by the request to `GetConfigurationService`. Remember that the service call is asynchronous. Because action depends on the configuration, it must go into the callback function of the service, because this code is executed after the service returns the configuration.

*Example 4-25 Updated Create Dossier action in DossierPlugin.js*

```
require(["dojo/_base/declare","dojo/_base/lang",
   "ecm/widget/dialog/AddContentItemDialog",
"ecm/model/Request","dossierPluginDojo/OpenDossierAction"],
function(declare, lang, AddContentItemDialog, Request, OpenDossierAction) {
lang.setObject("createDossierAction", function(repository, items) {
Request.invokePluginService("DossierPlugin",
   "GetConfigurationService",{
     requestCompleteCallback: function(response) {
   var dossierRootFolder = response.configuration[1].value;
   var dossierFolderClass =response.configuration[0].value;
   var templateDossierStructure =response.configuration[2].value;
   var _createFolderSubStructure = function(dossierFolder){
      var serviceParams = {
         icnRepository : repository.id,
         serverType : repository.type,
         dossierId : dossierFolder.id,
         templateFolderStructure: templateDossierStructure
      };
      Request.invokePluginService("DossierPlugin",
"CreateSubStructureService", {
         requestParams: serviceParams,
         requestCompleteCallback: function(response) {
            var newFolders = [];
            newFolders.push(dossierFolder);
            var openDossier = new OpenDossierAction();
            openDossier.performAction(repository, newFolders, null);
         }
      });
   };
   repository.retrieveItem(dossierRootFolder, function(dossierRootFolderItem)
{
      var addContentItemDialog = new AddContentItemDialog();
      addContentItemDialog.setDefaultContentClass(
dossierFolderClass);
      addContentItemDialog.show(repository, dossierRootFolderItem, false,
false, _createFolderSubStructure, null, false, null);
      addContentItemDialog.set("title","Create new Dossier");
      addContentItemDialog.setIntroText("This folder will be the top level
folder of your dossier.");
      });
     } //requestCompleteCallback
   });
  });
});
```

Also, the Open Dossier action must be adapted to read the configured folder class name; see Example 4-26.

*Example 4-26 Constructor of OpenDossierAction.js*

```
define(["dojo/_base/declare", "ecm/model/Action", "ecm/model/Request"],
   function(declare, Action, Request) {
   return declare("dossierPluginDojo.OpenDossierAction", [ Action ], {
   dossierFolderClass:null,
   isEnabled: function(repository, listType, items, teamspace,
resultSet) {
      var enabled = this.inherited(arguments);
      if (items && items[0].isFolder && items[0].getContentClass) {
         if (!this.dossierFolderClass) {
            Request.invokePluginService("DossierPlugin",
               "GetConfigurationService",
                  {
                     requestCompleteCallback: dojo.hitch(this,
                        function(response) {
                        this.dossierFolderClass =
                           response.configuration[0].value;
                     })
               });
         }
         var sameClass =
(items[0].getContentClass().name==this.dossierFolderClass);
         return enabled && items[0].isFolder() && sameClass;
      }
      return false;
   },
...
```

We add dossierFolderClass class property and, in the isEnabled method, we invoke the **GetConfigurationService** service with the requestCompleteCallback callback function to load the value from the configuration. Because the function's execution context in JavaScript is determined not when the function is designed but when it is executed, we must explicitly set the execution context to the object for which it is designed. With `dojo.hitch`, we can accomplish exactly this: We execute the function in the context of the OpenDossierAction object and are now able to set its dossierFolderClass property.

## 4.7 Dossier management in the real world

In this sample, and mainly for demonstration purposes, we create the substructure of the dossier through an implementation of the service extension point of IBM Content Navigator.

In a production environment, you would consider implementing the substructure in the back-end server. The general architecture guideline is to execute any task in the component for which it is best suitable. In our example, the creation of several subfolders is a typical back-end task and would be implemented in the back end rather than in the front end (such as IBM Content Navigator).

For IBM FileNet P8, this can be realized as an event action that is triggered when the top-level folder of the dossier is created. An associated event action handler can then asynchronously create the substructure.

Another simplification in the sample is that the design is to store the dossiers in a flat manner. Because we created only a few dossiers for demonstration purposes, this is not a problem. However, in a real-life scenario with maybe thousands or millions of customer dossiers, a poor design is to file all dossiers in one single parent folder. Consider managing the dossier in a substructure, for example, divided into regions, grouped by postal code, initial letters, and so on.

Finally, one consideration might be to eliminate physical folders in the repository and create a tree structure only at the client site where the documents are managed in a flat and unfiled manner at the repository. In this case, a tree node is a virtual folder and its content is determined by a search. This approach is shown in Chapter 5, “Building a custom repository search service” on page 173. A dossier constructed in this way is also called a *virtual dossier*.

## 4.8 Conclusion

This chapter describes how to implement and add a new plug-in to your IBM Content Navigator. Details are provided for developing most of the basic extension points of IBM Content Navigator such as PluginAction, PluginService, and PluginFeature. Configurative values for this plug-in are provided through the plug-in’s ConfigurationPane.

The example plug-in implements a simple dossier management extension for IBM Content Navigator. The base idea of dossiers is to get a structured view of the documents in your repository according to a primary order principle, such as all documents of a customer, all documents of an employee, and so on. It

enables users to create new dossiers and also search, open, and work on existing dossiers.

This plug-in is implemented for both IBM FileNet P8 and IBM Content Manager repositories.

5

# Building a custom repository search service

This chapter describes how to create an IBM Content Navigator plug-in to extend the base functionality of the product. It introduces how to create a custom repository search service and shows the results in existing ContentList widgets. With the custom repository search service, customers can make their own functions with search and view the results.

This chapter covers the following topics:

- Example overview
- Viewing results in ContentList widget
- Custom repository search service in sample plug-in
- Query string for search against repositories
- Creating a new plug-in with custom repository search service
- Adding a new function to the existing search service

© Copyright IBM Corp. 2012, 2014. All rights reserved.

## 5.1 Example overview

IBM Content Navigator provides SearchTemplate as the repository search model. If you want to extend IBM Content Navigator and create new functions, you can use the SearchTemplate model to build your own searches.

There are many reasons why you want to build your own searches. The current search template builder does not provide all of the complex Boolean operations that the search designer applet of IBM FileNet Workplace or IBM FileNet WorkplaceXT did at the time of writing. In addition, sometimes you want to search with a query string that is much easier to build than building an entire SearchTemplate model. Another reason might be that you do not want to build searches against the content management repositories; however, you want to build your search against a database. In this scenario, you still want to use the ContentList widget of Content Navigator to handle the results and objects. In these situations, a custom repository search service will be needed by users.

This does not mean you always need to use the custom repository search service when you extend IBM Content Navigator. When you need to handle saved searches in IBM Content Navigator, for example, retrieve saved searches and modify saved searches, SearchTemplate model must be used.

The example shown in this chapter is a plug-in of IBM Content Navigator that provides custom repository search service and shows the results in the existing ContentList widgets. You can use query string of IBM Content Manager or IBM FileNet Content Manager to do a search. This example extends the base functionality of IBM Content Navigator. With the custom repository search service, you can make your own functions with search and view the results.

## 5.2 Viewing results in ContentList widget

When you use the custom repository search service, you must consider how to handle and show the search results. ContentList widget, provided with IBM Content Navigator, is a powerful widget to show the search result set. It contains many modules that can be customized. We suggest using the ContentList widget whenever possible.

The ContentList widget looks like a general grid, which is the Dojo module that is used to display result set list. The ContentList is based on GridX packages. It is an extension of Dojo. There are several enhancements and customizable modules for it.

For example, there are three types of views of ContentList widget: ViewDetail, ViewMagazine, and ViewFilmStrip. You can decide which view will be used for your ContentList widget.

We show a ContentList example with ViewDetail, ViewMagazine, and ViewFilmStrip views modules configured. Three square icons are displayed on the top right button. Users can select which view they want to use. Figure 5-1 shows an example of ViewDetail view.

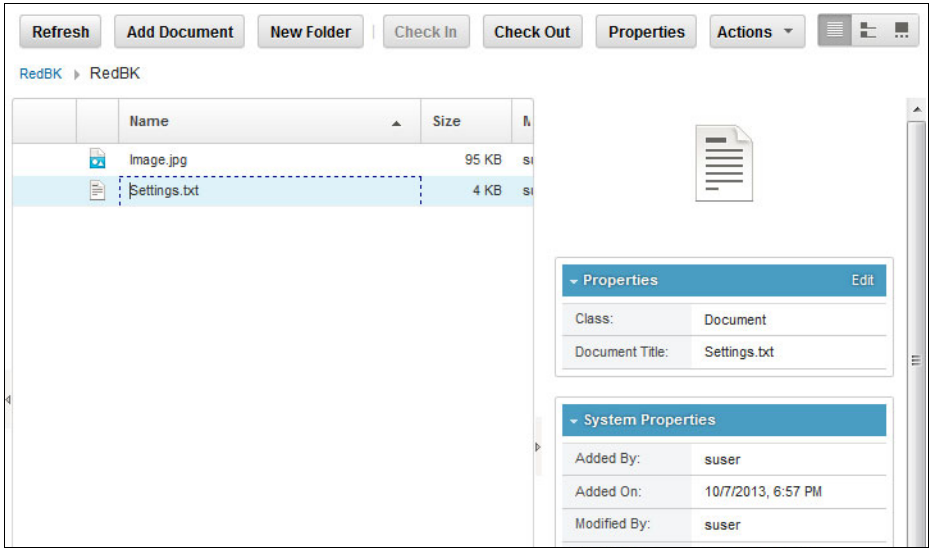



*Figure 5-1 ContentList widget with modules*

The toolbar module is at the top of the window. It shows several buttons. The breadcrumb module shows the path just under the toolbar module. Figure 5-1, shows **RedBK** → **RedBK**. The grid shows the document list with properties. The docinfo module shows a thumbnail and document properties on the right.

Figure 5-2 shows an example with the ViewMagazine view module.

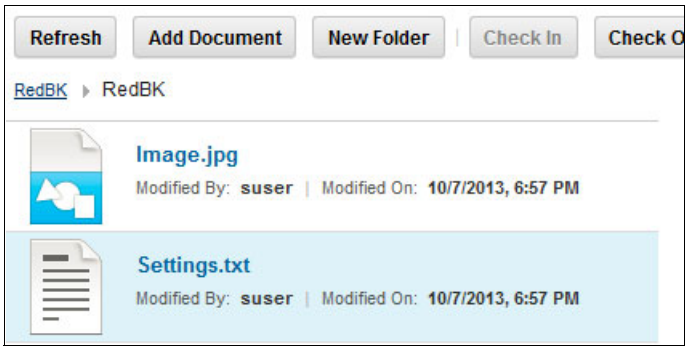



*Figure 5-2 Magazine view of ContentList*

Figure 5-3 shows a film strip view example. This affects only the grid.

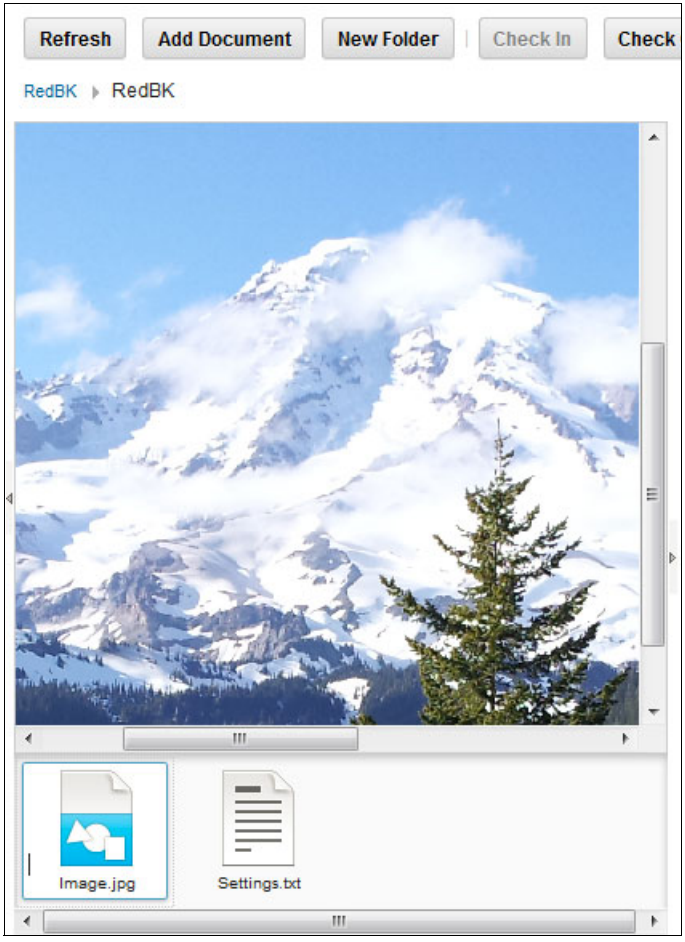



*Figure 5-3 FilmStrip of ContentList*

## ContentList Modules

The ContentList has several modules provided by IBM Content Navigator. You can choose and customize these modules within the ContentList.

- ContentList: The core module that shows data and launch actions.
- Toolbar: Provides a toolbar containing buttons.
- Bar: Provides the bar capability to arrange content list widgets.
- Breadcrumb: Provides the breadcrumb function.
- DocInfo: Shows document details.

- FilterData: Provides ability to filter data.
- InlineMessage: Shows messages inline.
- TotalCount: Shows the results of total count. It will be hidden for repositories that cannot provide the total count information.
- ViewDetail: A view module that shows detail view.
- ViewMagazine: A view module that shows magazine view.
- ViewFilmStrip: A view module that shows the filmstrip view.
- RowContextManu: Grid module that displays context menu.
- DndFromDesktopAddDoc: Grid module that allows users to drag-and-drop from desktop to grid.
- DndRowMoveCopy: A grid module that provides the ability to drag-and-drop a row. Then moves or copies when dropping the document or documents.
- DndRowCopy: A grid module that provides the ability to drag-and-drop a row, then copies when dropping the document or documents.
- DndDropOnly: A grid module that extends DndRowMoveCopy. It can disable the dragging of rows.

Example 5-1 is from the `SampleFeaturePane.js` sample plug-in file. It shows how to set the grid modules and other modules for ContentList widget. The DndRowMoveCopy and DndFromDesktopAddDoc modules provide the “drag and drop” functions. RowContextMenu module provides the context menu. You can create your own context menu module and replace the system menu. There is a setting in desktop configuration to control whether the ViewFilmStrip view can be used. The code here also honors this setting.

*Example 5-1 SampleFeaturePane.js*

```
getContentListGridModules: function() {
   var array = [];
   array.push(DndRowMoveCopy);
   array.push(DndFromDesktopAddDoc);
   array.push(RowContextMenu);
   return array;
},
getContentListModules: function() {
   var viewModules = [];
   viewModules.push(ViewDetail);
   viewModules.push(ViewMagazine);
   if (ecm.model.desktop.showViewFilmstrip) {
      viewModules.push(ViewFilmStrip);
   }
```

```
   var array = [];
   array.push(DocInfo);
   array.push({
      moduleClass: Bar,
      top: [
         [
            [
               {
                  moduleClass: Toolbar
               },
               {
                  moduleClasses: viewModules,
                  "className": "BarViewModules"
               }
            ]
         ]
      ]
   });
   return array;
},
```

To set the data to ContentList widget, use setResultSet (model, list Parent) method to set the result set. If the result set is from IBM Content Navigator model, such as SearchTemplate search result, the result can be set directly. If the result set is from other repository searches, such as a database, the result must be constructed along with the ResultSet model format. Then, ContentList can view it also.

## Helpful links

See the following sources for more information:

- Learn the basics of GridX:

  http://oria.github.io/gridx/

- Learn more about content list widgets package:

  http://pic.dhe.ibm.com/infocenter/cmgmt/v8r4m0/topic/com.ibm.developingeuc.doc/eucrf015.htm

## 5.3 Custom repository search service in sample plug-in

The custom repository search service is an extended service of IBM Content Navigator. IBM Content Navigator provides a sample in the sample plug-in.

Before implementing the custom repository search service, we first examine how it works. To see how it works, add the sample plug-in to your IBM Content Navigator. Configure a desktop to show the sample feature. Click **Save** to save the changes. Figure 5-4 shows adding of the sample feature to the desktop.

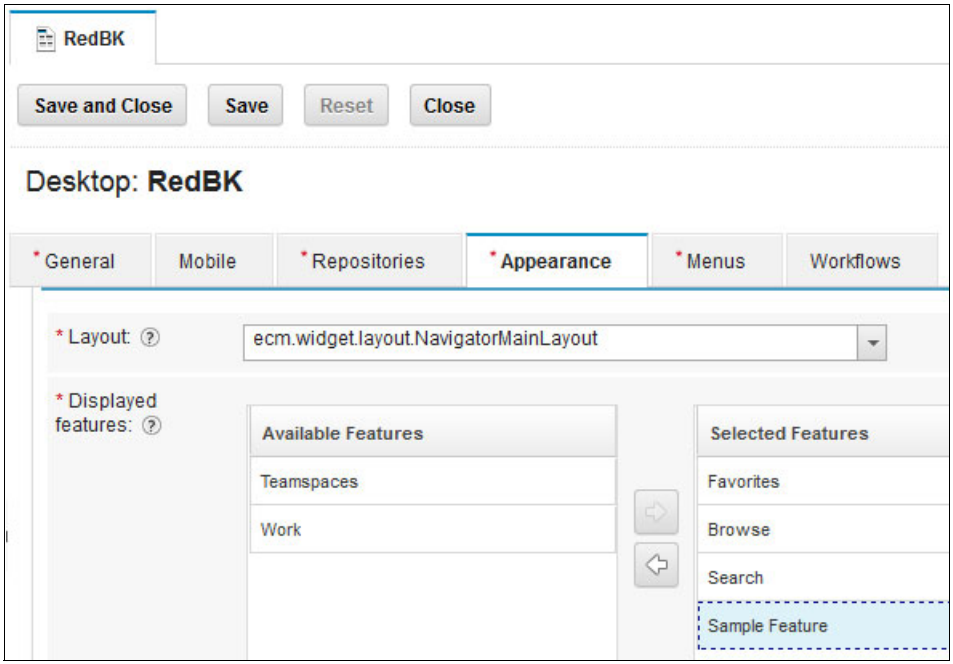



*Figure 5-4 Add sample features to display*

After you add the sample feature to the desktop, you can see it listed in the feature pane of the desktop. Click it; the view for custom repository search of the sample plug-in will be shown. The search service supports IBM Content Manager and IBM FileNet Content Manager. The prompt text for the sample plug-in search bar differs for the various repository types. The Sample Feature icon is the same as the original search feature. See Figure 5-5 on page 181.

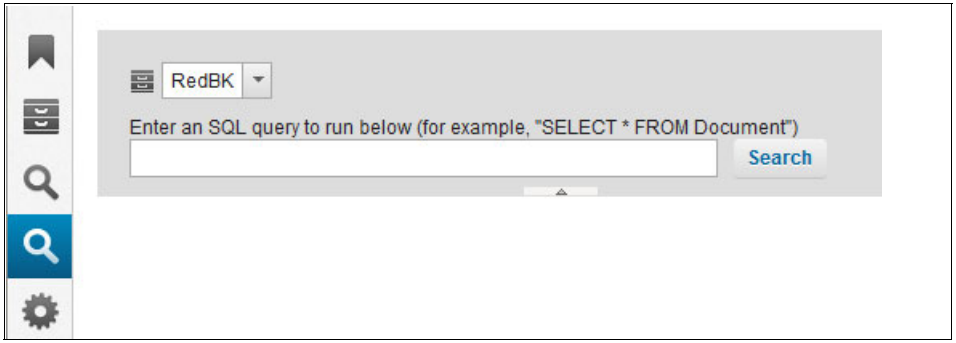



*Figure 5-5 Sample feature for custom repository search service*

To search, enter a valid query string and click **Search**. The search result is shown in the ContentList widget below. Be aware that the `infolder '/RedBk'` parameter means, in IBM FileNet Content Manager object store, that there is a folder named `RedBk` just under the root folder. Query string is described in 5.4, “Query string for search against repositories” on page 183.

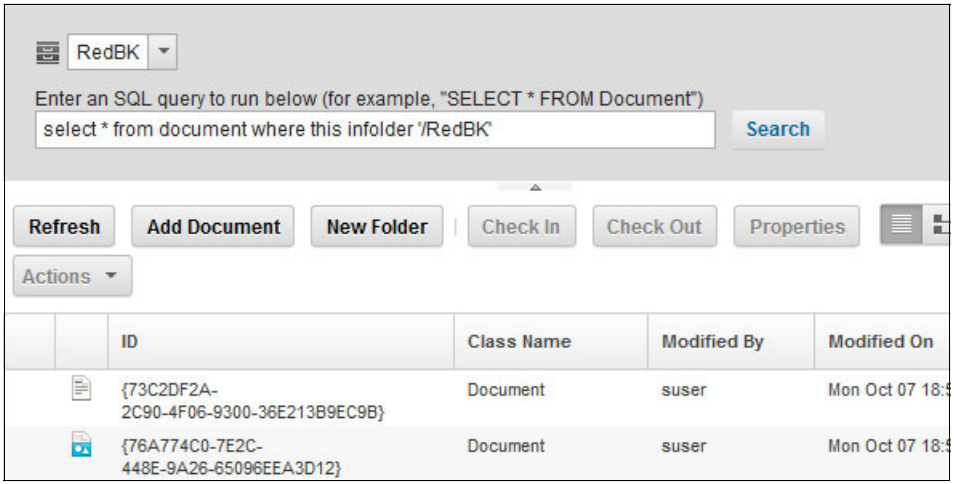



*Figure 5-6 Search with query string and show the results*

In this way, users can build their own search easily.

## 5.3.1 Sample code of custom repository search service

If you want to use the custom repository search service in some plug-in, start with the code in the sample plug-in first. If the function of the sample code meets your requirements exactly, you can use them as they are without any adjustment.

The sample code provides the ability to run query string against IBM Content Manager or IBM FileNet Content Manager repository.

In the sample plug-in project, there are three Java files related to custom repository search service:

- `SamplePluginSearchService.java`
- `SamplePluginSearchServiceCM.java`
- `SamplePluginSearchServiceP8.java`

SamplePluginSearchService.java is the real service that extends PluginService. The three methods in this file are as follows:

```
getId()
execute()
writeResponse()
```

The getId() method returns the serviced ID that will be used in Dojo code.

The execute() method carries the major functionality. It first gets the repository information from the request. Then it gets the query string. The callbacks.getSynchObject() method tries to get the synchronized repository object. If there is one, synchronize it, and perform a search. If there is none, just search directly. The getSynchObject() method is only for IBM Content Manger or IBM Content Manager OnDemand repositories. For IBM FileNet Content Manager repository, it always returns null. Then it writes the response.

The writeResponse() method writes response back, as its name indicates. It writes header for no-cache. Then it tries to compress the response if the client supports it or it sends the original JSON string.

`SamplePluginSearchServiceCM.java` and `SamplePluginSearchServiceP8.java` do the actual search, one for IBM Content Manager and one for IBM FileNet Content Manager. Both files build the response and are invoked from `SamplePluginSearchService.java`. When run, they do the following tasks:

1. Build the result structure.
2. Run the query string against the repository.
3. Get privilege masks for items.
4. Add attributes to the results.

The important part is how to build the result structure. In the methods buildCMReusltStructure() and buildP8ResultStructure(), users can see how to add columns for result set, and how to add columns for magazine view.

The SampleFeaturePane.js runSearch() method invokes samplePluginSearchService when users click the **Search** button.

The button-click event and Dojo method are connected in the `SampleFeaturePane.html` template. The result set is built in the Java service so it can be set to the ContentList widget with name as searchResults. The searchResults is defined in the `SampleFeaturePane.html` template.

Chapter 6, “Creating a feature with search services and widgets” on page 211 describes how to extend a new feature pane of IBM Content Navigator. It also explains the HTML template and other details.

## 5.4 Query string for search against repositories

The sample plug-in shows how to use the query string to search against repositories. In this section, we introduce the IBM Content Manager and IBM FileNet Content Manager query strings.

Like SQL is to a database, a query string is used to search in content management repositories. Different content management repositories have a different query string syntax.

### 5.4.1 IBM Content Manager query string

If the repository type is IBM Content Manager, compose query string by using the IBM Content Manager query language. It is not an SQL type language, but is more like XQuery, a language defined to search in XML format files. IBM Content Manager uses XPATH queries, which are transformed to SQL. This SQL statement will then be executed against the IBM Content Manager library server. The query string of IBM Content Manager can be tested in IBM Content Manager Client for Windows.

We show several examples next.

#### Sample IBM Content Manager query strings

The sample query string for IBM Content Manager in Example 5-2 looks up all items that have the `Title` attribute and contains the `book` string. The characters at the beginning of the line (`/*`) means to search in all ItemTypes, The forward slash (`/`) is used to qualify the ItemType scope. The at sign (`@`) is prefixed to IBM Content Manager internal attribute name in the rules. The square brackets (`[]`) are used to embrace the rules.

*Example 5-2 Search all items that have attribute “Title” contains the “book” string*

```
'/* [@Title like "%book%"]'
```

If you are going to search for items only in the ItemType `book`, the query string in Example 5-3 will have better performance.

*Example 5-3 Search for book items with "Title" attribute contains the "book" string*

```
'/BOOK [@Title like "%book%"]'
```

In IBM Content Manager, every item has a semantic type that is a system-defined attribute, The document's semantic type is 1 and the folder's semantic type is 2. If you want to search for only documents, the following rule must be joined to the rules with the `AND` keyword:

```
(@SEMANTICTYPE IN (1))
```

*Example 5-4 Search only documents with Title contains the "book" string*

```
'/* [(@SEMANTICTYPE IN (1)) AND @Title like "%book%"]'
```

The rule in Example 5-5 means that `icmadmin` is the last modifier.

*Example 5-5 Search for items that were last modified by "icmadmin"*

```
(@LASTCHANGEDUSERID = "icmadmin")
```

The query string for IBM Content Manager is flexible and powerful. It can get complicated. It is beyond the scope of this book to describe all in details. In this section, we only show some of the frequently used system attributes that can be joined to query rules. With these examples, you can try some simple searches.

## Helpful links

For more information about query string for IBM Content Manager, see the following sources for more information:

- Searching for data:

  http://pic.dhe.ibm.com/infocenter/cmgmt/v8r5m0/index.jsp?topic=%2Fcom.ibm.programmingcm.doc%2Fdcmaq009.htm

- System tables, ICMUT00300001 (Base table):

  http://bit.ly/1goG3aN

## 5.4.2 IBM FileNet Content Manager query string

Generally, the IBM FileNet Content Manager query string syntax conforms to SQL-92 standard. When it is running, it is transformed to SQL against the database.

There are specific IBM FileNet Content Manager parameters. For example, if you want to search for items in a particular folder, then the *infolder* parameter must be used. The query string in Example 5-6 searches for all documents with all their properties in the folder `RedBK` called under the root folder.

**Note:** If you copy this query string from the PDF file, replace the open and close single quotation marks (**‘ ’**) characters to standard single quotation marks on your web page.

*Example 5-6 Search all documents in the “RedBk” folder and return all their attributes*

```
select * from document where this infolder '/RedBK'
```

The Query Builder within the Enterprise Manager administration client of IBM FileNet Content Manager provides the function to build and run the query strings. You can use it to test whether your query string works and the results you will get. Query Builder can be launched from the Windows system that installed IBM FileNet Content Manager Enterprise Manager administration client.

To test your query string with Query Builder of the Enterprise Manager administration client, use the following steps:

1. Start Enterprise Manager for IBM FileNet Content Manager.
2. Go to the object store that you want to search.
3. Right-click on **Search Results** node and select **New Search**.

   You can also open a saved search to run or make adjustments on an existing search. To see the saved searches, select **View** → **SQL View.**
4. Write your query string. Click **OK** to determine if the syntax is correct and get the search results.

There is a difference between the query strings running in Enterprise Manager and the those running with API. The query strings that run in Enterprise Manager must use `This` or `TableName`. This as the first value. See Figure 5-7 on page 186.

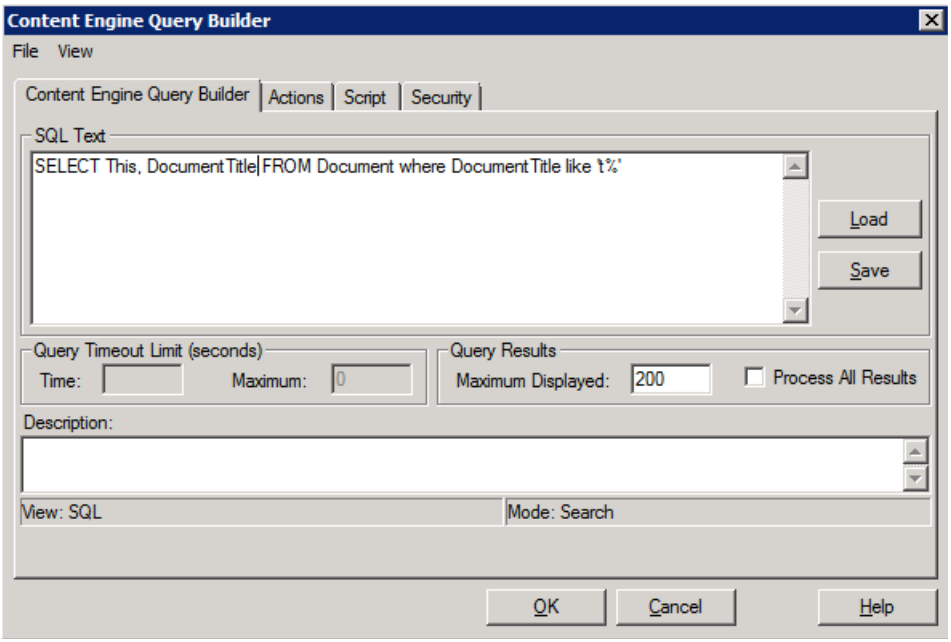



*Figure 5-7 Sample query string running Enterprise Manager Query Builder*

Using Query Builder, you can try your query string first to avoid debugging in the application. That will save a lot of time.

For IBM FileNet Content Manager Version 5.2 or later, there is an administration client called Administration Console for Content Platform Engine (ACCE). To access it, use the following URL through your browser:

```
http://<CEServer>:<Port>/acce
```

To test your query string from the Administration Console for Content Platform Engine, complete the following steps:

1. Launch the browser client.
2. Select the object store you want to access in Object Stores list.
3. Select **Search** node.
4. Construct a search in the Simple Search tab or run a query string in the SQL Query tab. See Figure 5-8 on page 187.

In ACCE, the search result returns only total quantity for running the query string. Test your query string here to avoid syntax error.

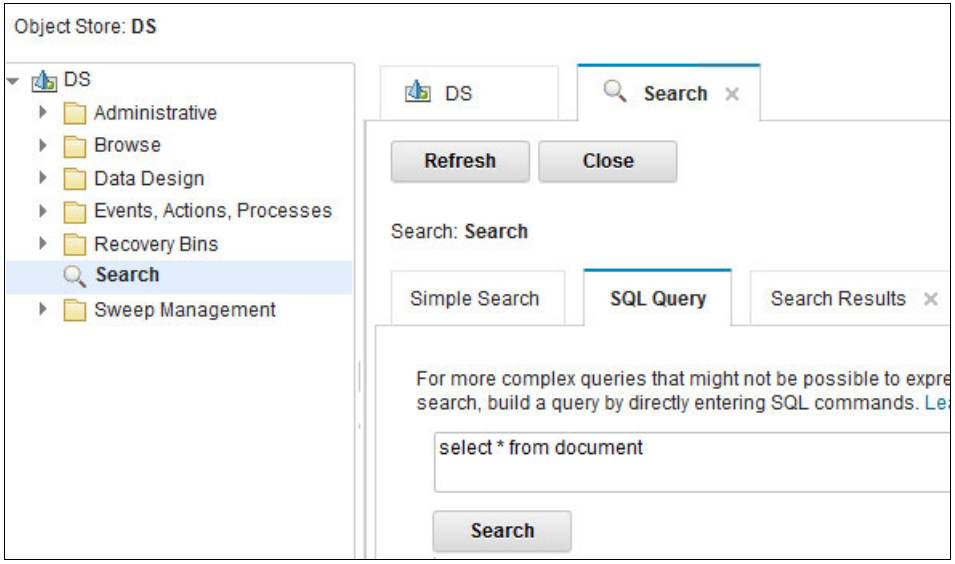



*Figure 5-8 Search with query string in ACCE*

### Helpful link

To learn more about ACCE and finding objects, see the IBM Knowledge Center:

`http://pic.dhe.ibm.com/infocenter/p8docs/v5r2m0/index.jsp?topic=%2Fcom.ibm.p8.ce.admin.tasks.doc%2Fp8pcc187.htm`

## 5.5 Creating a new plug-in with custom repository search service

If you want to create a new plug-in to contain the custom repository search, the easiest way is to use the existing code in the sample plug-in if possible. In this section, we show you how to add the existing custom repository search service into the plug-in.

### 5.5.1 Creating a new plug-in project

Chapter 3, “Setting up the development environment” on page 73 describes how to start a new plug-in project with the Eclipse plug-in. Follow the steps described there to create a new plug-in named CustomSearchPlugin. The configuration is shown in Figure 5-9 on page 188.

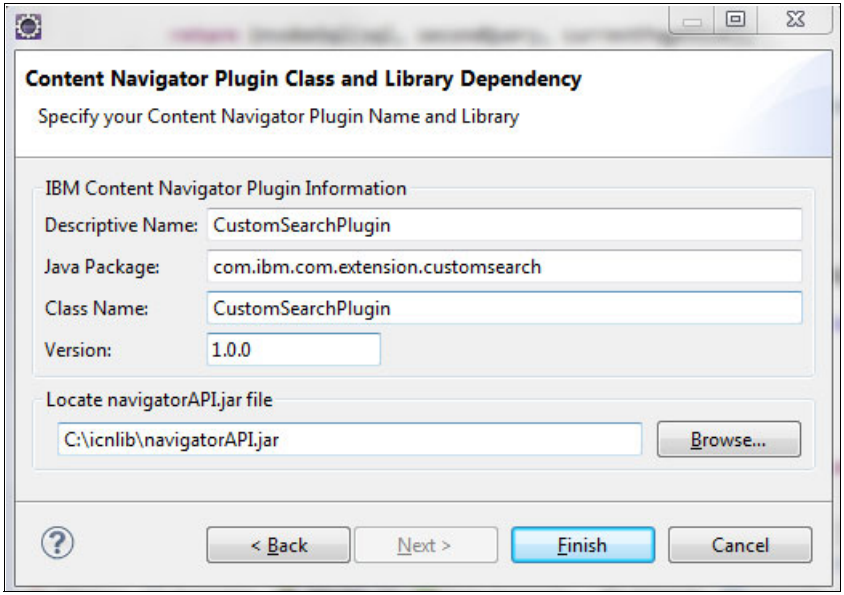



*Figure 5-9 The new plug-in, CustomSearchPlugin configuration*

## 5.5.2 Importing search service from sample plug-in

After the plug-in project is created, create the search service in the CustomSearchPlugin project. Follow these steps:

1. Expend the **CustomSearchPlugin** project in Eclipse and expand the **src** node.
2. Right-click the **com.ibm.com.extension.customsearch** package and select **IBM Content Navigator** → **Server Extensions** → **New Service**.

3. For the class name field, enter `SearchService`. See Figure 5-10.

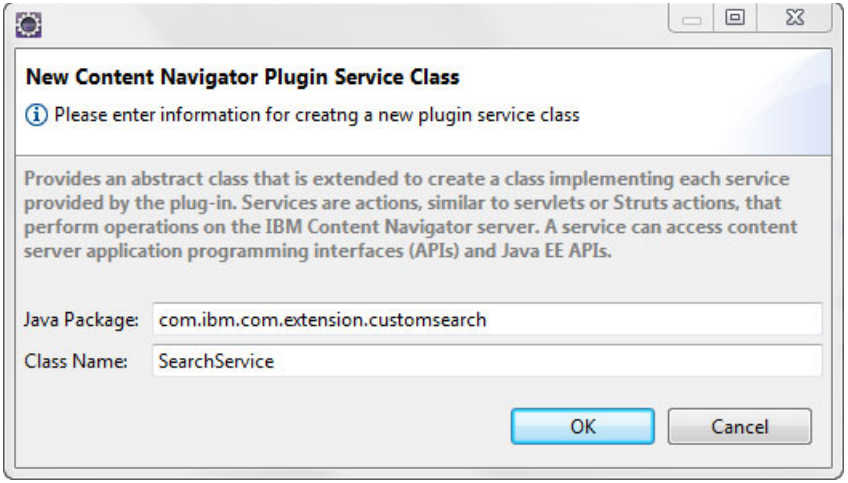



*Figure 5-10 Enter the class name for the new service*

4. Click **OK**. Now `SearchService.java` is created. It is also registered into the `CustomSearchPlugin.java` file as a service. It does not yet contain any function.
5. Copy the following JAR files to the directory where you keep JAR files related to Content Navigator, for example, where you put `navigatorAPI.jar`. Add them into the build path of CustomSearchPlugin project.

   ```
   jace.jar
   struts-1.1.jar
   cmbicmsdk81.jar
   ```

   The `jace.jar` file provides the Java API for IBM FileNet Content Manager. The `cmbicmsdk81.jar` file provides the Java API for IBM Content Manager. The `struts-1.1.jar` file provides struts-related API.

   You can find them if the sample plug-in project is set up properly as described in Chapter 3, “Setting up the development environment” on page 73.
6. Copy the following Java files into the com.ibm.com.extension.customsearch package:

   ```
   SamplePluginSearchServiceCM.java
   SamplePluginSearchServiceP8.java
   ```

   If you encounter a compile error for these two Java files, check that the `jace.jar`, `struts-1.1.jar`, and `cmbicmsdk81.jar` files are in the project build path.

7. Invoke the search service logic into `SearchService.java`. Do this by copying the code from the `SamplePluginSearchService.java` file:
   - Copy the definitions of REPOSITORY_ID, REPOSITORY_TYPE, and QUERY to `SearchService.java`.
   - Copy the execute() method and writeResponse() method from `SamplePluginSearchService.java` to `SearchService.java`. Make sure to remove the old empty execute() method from `SearchService.java`.
8. There might be an import error that points to the sample plug-in path of the following two files. Remove the two import lines.

   ```
   SamplePluginSearchServiceCM.java
   SamplePluginSearchServiceP8.java
   ```

The code in the `SearchService.java` file is similar to Example 5-7.

*Example 5-7 SearchService.java*

```
package com.ibm.com.extension.customsearch;

import java.io.OutputStreamWriter;
import java.io.Writer;
import java.util.zip.GZIPOutputStream;

import javax.security.auth.Subject;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.filenet.api.util.UserContext;
import com.ibm.ecm.extension.PluginService;
import com.ibm.ecm.extension.PluginServiceCallbacks;
import com.ibm.ecm.json.JSONMessage;
import com.ibm.ecm.json.JSONResultSetResponse;

/**
 * Provides an abstract class that is extended to create a class
 * implementing each service provided by the plug-in. Services are
 * actions, similar to servlets or Struts actions, that perform
 * operations on the IBM Content Navigator server. A service can access
 * content server application programming interfaces (APIs) and Java EE
 * APIs.
 * <p> Services are invoked from the JavaScript functions that are
 * defined for the plug-in by using the
 * <code>ecm.model.Request.invokePluginService</code> function.
 * </p>
 * Follow best practices for servlets when implementing an IBM Content
 * Navigator plug-in service. In particular, always assume
 * multi-threaded use and do not keep unshared information in instance
* variables.
```

```
 */
public class SearchService  extends PluginService {

   public static final String REPOSITORY_ID = "repositoryId";
   public static final String REPOSITORY_TYPE = "repositoryType";
   public static final String QUERY = "query";
   /**
    * Returns the unique identifier for this service.
    * <p>
    * <strong>Important:</strong> This identifier is used in URLs so it must
    * contain only alphanumeric characters.
    * </p>
    *
    * @return A <code>String</code> that is used to identify the service.
    */
   public String getId() {
      return "SearchService";
   }

   /**
    * Returns the name of the IBM Content Navigator service that this
    * service overrides. If this service does not override an IBM
    * Content Navigator service, this method returns <code>null</code>.
    *
    * @returns The name of the service.
    */
   public String getOverriddenService() {
      return null;
   }

   /**
    * Performs the action of this service.
    *
    * @param callbacks
    * An instance of the <code>PluginServiceCallbacks</code>
    * class that contains several functions that can be used by the
    * service. These functions provide access to the plug-in
    * configuration and content server APIs.
    * @param request
    * The <code>HttpServletRequest</code> object that provides the
    * request. The service can access the invocation parameters from
    * the request.
    * @param response
    * The <code>HttpServletResponse</code> object that is generated
    * by the service. The service can get the output stream and
    * write the response. The response must be in JSON format.
    * @throws Exception
    * For exceptions that occur when the service is running. If the
    * logging level is high enough to log errors, information about
```

```
     * the exception is logged by IBM Content Navigator.
     */
    public void execute(PluginServiceCallbacks callbacks,
          HttpServletRequest request, HttpServletResponse response)
          throws Exception {String methodName = "execute";
          callbacks.getLogger().logEntry(this, methodName, request);

          String repositoryId = request.getParameter(REPOSITORY_ID);
          String repositoryType = request.getParameter(REPOSITORY_TYPE);
          String query = request.getParameter(QUERY);

          JSONResultSetResponse jsonResults = new JSONResultSetResponse();
          jsonResults.setPageSize(350);

          try {
             if (repositoryType.equals("p8")) {
                Subject subject = callbacks.getP8Subject(repositoryId);
                UserContext.get().pushSubject(subject);
             }

             Object synchObject = callbacks.getSynchObject(repositoryId,
repositoryType);
             if (synchObject != null) {
                synchronized (synchObject) {
                   if (repositoryType.equals("cm")) {

SamplePluginSearchServiceCM.executeCMSearch(repositoryId, query, callbacks,
jsonResults, request.getLocale());
                   } else if (repositoryType.equals("p8")) {

SamplePluginSearchServiceP8.executeP8Search(repositoryId, query, callbacks,
jsonResults, request.getLocale());
                   }
                }
             } else {
                if (repositoryType.equals("cm")) {
                   SamplePluginSearchServiceCM.executeCMSearch(repositoryId,
query, callbacks, jsonResults, request.getLocale());
                } else if (repositoryType.equals("p8")) {
                   SamplePluginSearchServiceP8.executeP8Search(repositoryId,
query, callbacks, jsonResults, request.getLocale());
                }
             }

             // Write results to response
             writeResponse(request, response, jsonResults);

          } catch (Exception e) {
             // provide error information
```

```
            callbacks.getLogger().logError(this, methodName, request, e);

            JSONMessage jsonMessage = new JSONMessage(0, e.getMessage(), "This
error may occur if the search string is invalid.", "Ensure the search string is
the correct syntax.", "Check the IBM Content Navigator logs for more details.",
"");
            jsonResults.addErrorMessage(jsonMessage);
            writeResponse(request, response, jsonResults);
         } finally {
            if (repositoryType.equals("p8")) {
               UserContext.get().popSubject();
            }

            callbacks.getLogger().logExit(this, methodName, request);
         }
      }

      private void writeResponse(HttpServletRequest request,
HttpServletResponse response, JSONResultSetResponse json) throws Exception {
         Writer writer = null;

         try {
            // Prevent browsers from returning cached response on subsequent
request
            response.addHeader("Cache-Control", "no-cache");

            // GZip JSON response if client supports it
            String acceptedEncodings = request.getHeader("Accept-Encoding");
            if (acceptedEncodings != null && acceptedEncodings.indexOf("gzip")
>= 0) {
               if (!response.isCommitted())
                  response.setBufferSize(65536); // since many times response
is larger than default buffer (4096)
               response.setHeader("Content-Encoding", "gzip");
               response.setContentType("text/plain"); // must be text/plain
for firebug
               GZIPOutputStream gzos = new
GZIPOutputStream(response.getOutputStream());
               writer = new OutputStreamWriter(gzos, "UTF-8");
               // Add secure JSON prefix
               writer.write("{}&&");
               writer.flush();
               json.serialize(writer);
            } else {
               response.setContentType("text/plain"); // must be text/plain
for firebug
               response.setCharacterEncoding("UTF-8");
               writer = response.getWriter();
               // Add secure JSON prefix
```

```
                writer.write("{}&&");
                writer.flush();
                json.serialize(writer);
             }
          } catch (Exception e) {
             throw e;
          } finally {
             if (writer != null)
                writer.close();
          }
       }
    }
```

If you try to build the CustomSearchPlugin project with `build.xml`, it will fail. You must add the JAR file path to the `build.xml` file. The classpath section of `build.xml` is shown in Example 5-8.

*Example 5-8 Class path section of build.xml*

```
<path id="classpath">
   <pathelement location="C:/icnlib/navigatorAPI.jar" />
   <pathelement location="C:/icnlib/jace.jar" />
   <pathelement location="C:/icnlib/struts-1.1.jar" />
   <pathelement location="C:/icnlib/cmbicmsdk81.jar" />
   <pathelement location="./lib/j2ee.jar" />
   <pathelement location="./temp" />
</path>
```

Now, your CustomSearchPlugin project has the custom repository search service from the sample plug-in. You must test it. To do so, import the feature pane from the sample plug-in.

### 5.5.3 Importing feature pane from sample plug-in

Create a new feature in the CustomSearchPlugin project:

1. Expand the project in Eclipse and expand the **src** node.
2. Right-click on the `com.ibm.com.extension.customsearch` package.
3. Select **IBM Content Navigator** → **New Feature**.
4. Enter the configurations for the new plug-in feature. See Figure 5-11 on page 195.

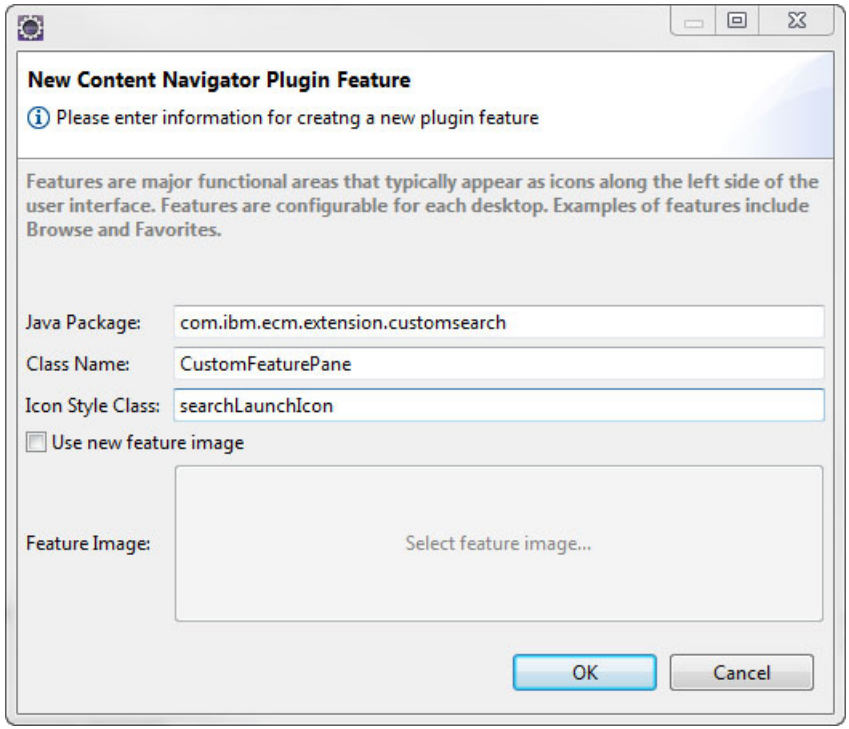



*Figure 5-11 Enter configuration for the new plug-in feature*

In Figure 5-11, the Icon Style Class is the class that shows the icon of the new feature. We use searchLaunchIcon class, which is what SamplePluginFeature uses. It is provided by IBM Content Navigator definitions. You can build your own class with custom icon for the new feature. Or you can click **Select feature image** to select the icon file. If you use myIconClass as the class name here, a class should be defined in `CustomSearchPlugin.css` file, as in Example 5-9.

*Example 5-9 Icon class example*

```
.myIconClass {
   width: 32px;
   height: 32px;
   background: url('images/<NameOfYourImage>.png') no-repeat;
}
```

To learn how the icon is set and how to add more resources, see Chapter 6, “Creating a feature with search services and widgets” on page 211.

Now we get a new empty feature. You can test it by building and deploying it to IBM Content Navigator to be sure everything works successfully.

Check your CustomSearchPlugin project to see what happened when creating a new feature. Unlike extending a service, several files are created:

```
CustomFeaturePane.java
CustomFeaturePane.js
CustomFeaturePane.html
```

A new feature usually has its own pane and template. See Figure 5-12.

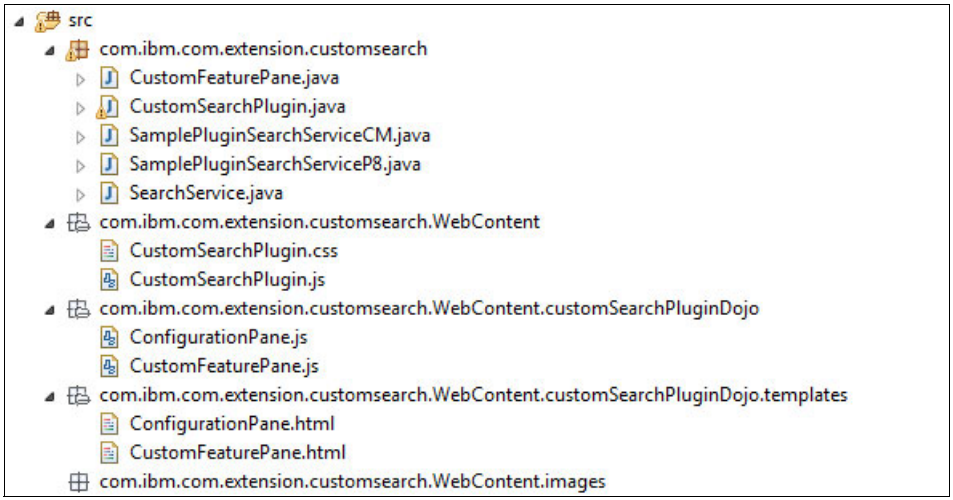



*Figure 5-12 Structure after creating a new feature*

Now you can move the feature-related code from the sample plug-in to the new feature:

1. For the `CustomFeaturePane.html` file, replace its content with the content of `SamplePluginFeaturePane.html` in the sample plug-in.
2. For the `CustomFeaturePane.js` file, replace its content with the content of `SamplePluginFeaturePane.js`.
3. Make additional adjustments to the `CustomFeaturePane.js` file as follows:
   - Modify the last line of the define sector as follows:

     `dojo/text!./templates/CustomFeaturePane.html`

     The reason for modifying it is because the template we use is `CustomFeaturePane.html` in this project, but the line in `SamplePluginFeaturePane.js` points to a different one.

– Modify the first value of declare as follows:

  `customSearchPluginDojo.CustomFeaturePane`

– Find the `Request.invokePluginService` line in the runSearch() method and change it to the following text:

  ```
  Request.invokePluginService("CustomSearchPlugin",
  "SearchService",
  ```

  The first value is plug-in ID defined in the CustomSearchPlugin.getId() method. The second value is the search service ID defined in SearchService.getId(). This is how the service is invoked.

## 5.5.4 Building and deploying the new plug-in

After moving the search service, and the pane to use it, from the sample plug-in to another plug-in project, you can build and deploy the new plug-in.

To build and deploy the new plug-in, follow these steps:

1. Find the `build.xml` in your project, right-click on it, and select **Run As** → **Ant Build**. If a compile error occurs, check that all JAR files in the project build path are in the `build.xml` build path.

   When the build is done, a file named `CustomSearchPlugin.jar` is generated in the plug-in project. Refresh in Eclipse to see it.

2. Go to IBM Content Navigator Administration Desktop using the following URL:

   `http://`*`NavigatorURL`*`:`*`port`*`/navigator?desktop=admin`

3. Click the **Plug-ins** tab, select **New Plug-in**, enter the `CustomSearchPlugin.jar` path, and click **Load**.

Information about the plug-in is displayed (Figure 5-13).

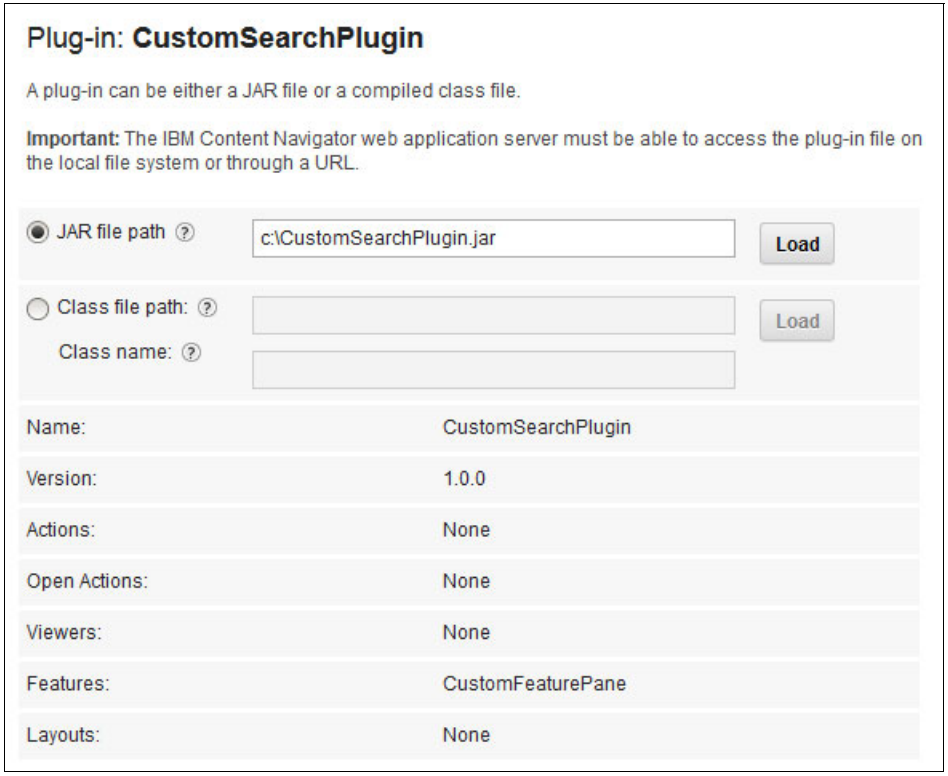



*Figure 5-13 Add custom search plug-in Administration Desktop*

4. Open the desktop you want to configure, click the **Appearance** tab, and add `CustomFeaturePane` to the Selected Features. See Figure 5-14.

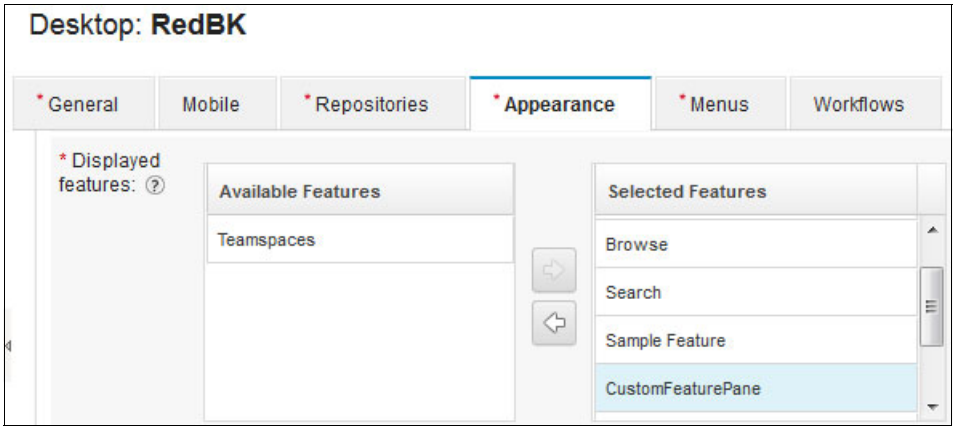



*Figure 5-14 Add CustomFeaturePane to Selected Features*

5. Test the new feature by again accessing the desktop of IBM Content Navigator in the browser:

   `http://`*`NavigatorURL`*`:`*`port`*`/navigator?desktop=RedBK`

   If you can see the sample plug-in feature pane, then it is successful.

# 5.6 Adding a new function to the existing search service

Now that we get the custom repository search service as the sample plug-in, it is time to do more customization.

This section covers the enhancements that are done for IBM FileNet Content Manager only in the custom search plug-in sample.

## 5.6.1 Adding a paging function

The `SamplePluginSearchServiceP8.java` code has a parameter defined for page size:

```
int pageSize = 350;
```

The code in the original `SamplePluginSearchServiceP8.java` retrieves the first page of the search result. Therefore, with this search service, you get the first 350 items of the results from IBM FileNet Content Manager. In the IBM Content Navigator browse pane or search pane, you can scroll the ContentList to the

bottom to trigger the continued search for the next few pages. We add this feature to the custom search plug-in. Again, this implementation is done for IBM FileNet Content Manager repository only.

Figure 5-15 shows when ContentList triggers paging search, there will be some lines with blue columns to show that they are waiting for results.

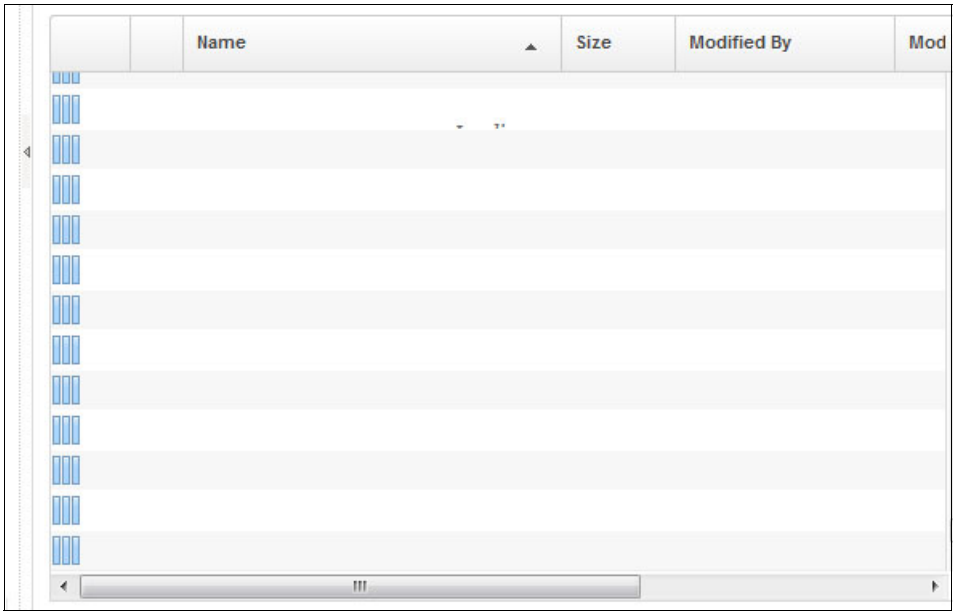



*Figure 5-15 ContentList in paging search*

IBM FileNet Content Manager API provides the paging ability by the PageIterator object. In the original `SamplePluginSearchServiceP8.java` in the sample plug-in, the code shown in Example 5-10 gets the first page from the search result. You can find the complete code in the sample plug-in.

*Example 5-10 Code snippet from SamplePluginSearchServiceP8.java*

```
// Retrieve the first pageSize results.
       int pageSize = 350;
       List<Object> searchResults = new ArrayList<Object>(pageSize);
       IndependentObjectSet resultsObjectSet = searchScope.fetchObjects(searchSQL,
pageSize, filter, true);
       PageIterator pageIterator = resultsObjectSet.pageIterator();
       if (pageIterator.nextPage()) {
          for (Object obj : pageIterator.getCurrentPage()) {
             searchResults.add(obj);
          }
       }
```

To get the next page of the search result, use the pageIterator.nextPage() method. The method to implement the paging is to save the pageIterator object, then use it to get the next page when users trigger the continued search.

The searches are divided into two types:

- Start the search, get the first page of the results, save the pageIterator object into the session. Put the continuationData value into the ResultSet.
- Get the pageIterator from session. If it exists, get the next page of the search result from it.

The first type of search can be implemented by modifying the `SamplePluginSearchServiceP8.java`. We add the pageIterator object into session. In the custom search plug-in `SamplePluginSearchServiceP8.java` file final version, the code includes this. When the first page items count equals pageSize, it means there might be more results. The pagerIterator sessionKey is put in the continuationData result set. Then the pageIterator is put into session with that sessionKey as the name. Example 5-11 shows the key snippet of the code. Check the complete code in `SamplePluginSearchServiceP8.java` in the CustomSearchPlugin project for this chapter. The itemCount parameter is the result quantity from the first page. The request parameter is the request object of web application.

*Example 5-11 Set key and pageIterator into session*

```
String sessionKey = "pagerIterator";
request.getSession().removeAttribute(sessionKey);
if (itemCount == pageSize) {
   jsonResultSet.put("continuationData", sessionKey);
   //this require CE version >=5.0
   request.getSession().setAttribute(sessionKey, pageIterator);
}
```

The ContentList widget already provides the paging function required part.

When you scroll through the ContentList, if the ResultSet object contains the continuationData parameter, then the ContentList will trigger a service such as `/p8/continueQuery`. The first path value depends on the repository type. It might be `/cm/continueQuery` or `/cmis/continueQuery`. This service can return a next page if there is one.

The service that the ContentList triggers is the IBM Content Navigator service. It processes the paging search in a different way than what we described here. So you need your own for the custom search plug-in to do what you want.

To trigger your own service, use the request filter feature that is included with IBM Content Navigator. Request filter is a mechanism that you can use to replace any request with your own. That means when ContentList sends a request to `/p8/continueQuery`, you can capture it and use another method to response to it.

With IBM Content Navigator, you can more easily extend and build a request filter. We build a request filter named `ContinueQueryService.java` to replace the `/p8/continueQuery` for the plug-in.

Complete the following steps:

1. Create a request filter by using Eclipse plug-in. Right-click on the packages in the custom search plug-in project, and select **IBM Content Navigator** → **Server Extensions** → **New Request Filter**.
2. In the new request filter dialog, configure it as follows (see Figure 5-16 on page 203):
   - Enter the Class Name as `ContinueQueryService`.
   - Select **/p8/continueQuery** from the services list and click **Append selected**. You can select more than one services to be filtered. For this sample, we modify the code only for IBM FileNet Content Manager.
3. Click **OK**. The request filter is then generated and registered into `CustomSearchPlugin.java`.

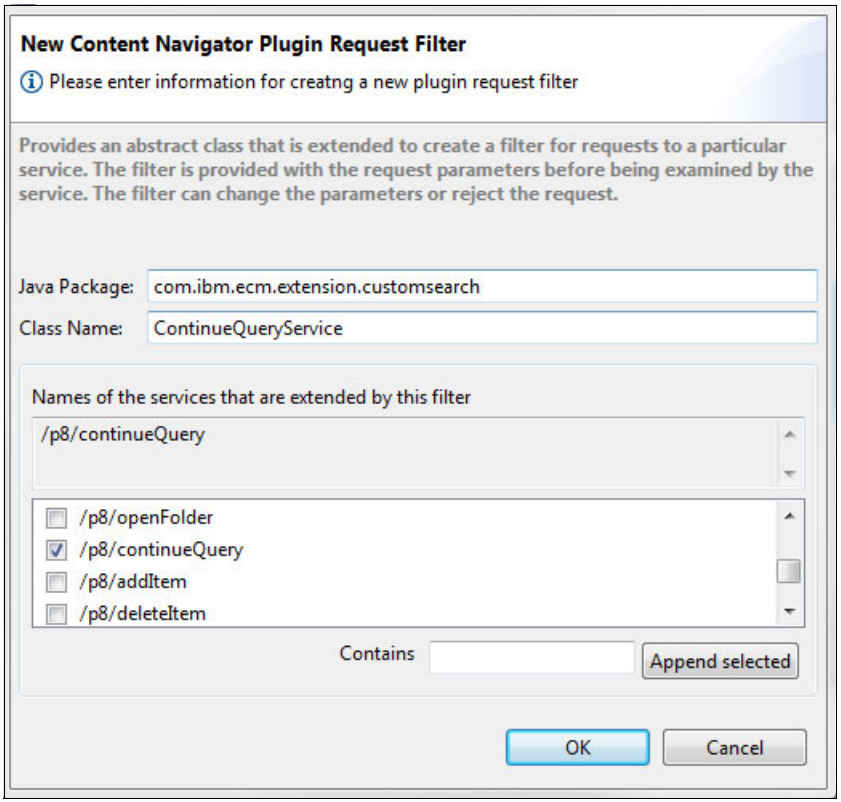



*Figure 5-16 Configure a new request filter*

In the new `ContinueQueryService.java`, the getFilteredServices() method (Example 5-12) shows which service is filtered. It can be a list.

*Example 5-12 getFilteredServices method in ContinueSearchService.java*

```
@Override
public String[] getFilteredServices() {
   return new String[] { "/p8/continueQuery" };
}
```

The core method is in the filter() method. The return type should be JSONObject. You can add any code you want in the method.

We try to get the continuationData string from the request, and then use it as the key to get the value from session. If the value is not null, it should be the pageIterator we put into the session before. Then, we will use it to get the next page of the search result. If it is null, that means the request is not for the custom

search plug-in, but for IBM Content Navigator. In this situation, return null. It will go to the original IBM Content Navigator service handler. See Example 5-13.

*Example 5-13 filter method in ContinueQueryService.java*

```
@Override
   public JSONObject filter(PluginServiceCallbacks callbacks,
HttpServletRequest request, JSONArtifact jsonRequest) throws Exception
{
      String continueQeurySessionKey =
request.getParameter("continuationData");
      if (request.getSession().getAttribute(continueQeurySessionKey) !=
null) {
         //the request is from plug-in service
         JSONResultSetResponse jsonResults = new
JSONResultSetResponse();
         this.execute(callbacks, request, jsonResults);
         return jsonResults;
      } else {
         //the request is not related with sample plug-in search
service,
         //goto the default ICN service handler.
         return null;
      }
   }
```

The logic to get the next page data and return the result is in method execute(). The method gets the pageIterator from session with the key we created. If the pageIterator is not null, then use pageIterator to get the next page of search result. See Example 5-14. We modify the pageSize to 50 because testing the paging feature with this value is easier. Although you can keep the original value 350 if you want, you will need a search result larger than 350 to see if the paging works.

*Example 5-14 execute method in ContinuQueryService.java*

```
public static final String REPOSITORY_ID = "repositoryId";
public static final String REPOSITORY_TYPE = "repositoryType";
public static final int pageSize = 50;

public void execute(PluginServiceCallbacks callbacks, HttpServletRequest
request, JSONResultSetResponse jsonResults) throws Exception {
      String methodName = "execute";
      callbacks.getLogger().logEntry(this, methodName, request);

      String repositoryId = request.getParameter(REPOSITORY_ID);
```

```
      String continueQeurySessionKey =
request.getParameter("continuationData");

      jsonResults.setPageSize(pageSize);
      List<Object> searchResults = new ArrayList<Object>(pageSize);
      int itemCount = 0;
      try {
         Subject subject = callbacks.getP8Subject(repositoryId);
         UserContext.get().pushSubject(subject);

         Object synchObject = callbacks.getSynchObject(repositoryId, "p8");
         if (synchObject != null) {
            synchronized (synchObject) {
               PageIterator pageIterator = (PageIterator)
request.getSession().getAttribute(continueQeurySessionKey);
               if (pageIterator.nextPage()) {
                  for (Object obj : pageIterator.getCurrentPage()) {
                     searchResults.add(obj);
                     itemCount++;
                  }
               }
               if (itemCount == pageSize) {
                  String sessionKey = "pagerIterator";
                  jsonResults.put("continuationData", sessionKey);
               } else {

request.getSession().removeAttribute(continueQeurySessionKey);
               }
            }
         } else {
            PageIterator pageIterator = (PageIterator)
request.getSession().getAttribute(continueQeurySessionKey);
            if (pageIterator.nextPage()) {
               for (Object obj : pageIterator.getCurrentPage()) {
                  searchResults.add(obj);
                  itemCount++;
               }
            }
            if (itemCount == pageSize) {
               String sessionKey = "pagerIterator";
               jsonResults.put("continuationData", sessionKey);
            } else {
               request.getSession().removeAttribute(continueQeurySessionKey);
            }
         }
         // Retrieve the privilege masks for the search results.
         HashMap<Object, Long> privMasks =
callbacks.getP8PrivilegeMasks(repositoryId, searchResults);
         ObjectStore objectStore = callbacks.getP8ObjectStore(repositoryId);
```

```
            for (Object searchResult : searchResults) {
               Document doc = (Document) searchResult;
               /*
                *  IDs use the form:
                *  <object class name>,<object store ID>,<object ID>
                */
               StringBuffer sbId = new StringBuffer();

sbId.append(doc.getClassName()).append(",").append(objectStore.get_Id().toStrin
g()).append(",").append(doc.get_Id().toString());

               long privileges = (privMasks != null) ? privMasks.get(doc) : 0L;

               JSONResultSetRow row = new JSONResultSetRow(sbId.toString(),
doc.get_Name(), doc.get_MimeType(), privileges);

               // Add locked user information (if any)
               row.addAttribute("locked", doc.isLocked(),
JSONResultSetRow.TYPE_BOOLEAN, null, (new Boolean(doc.isLocked())).toString());
               row.addAttribute("lockedUser", doc.get_LockOwner(),
JSONResultSetRow.TYPE_STRING, null, doc.get_LockOwner());
               row.addAttribute("currentVersion", doc.get_IsCurrentVersion(),
JSONResultSetRow.TYPE_BOOLEAN, null, (new
Boolean(doc.get_IsCurrentVersion())).toString());

               // Add the attributes
               row.addAttribute("ID", doc.get_Id().toString(),
JSONResultSetRow.TYPE_STRING, null, doc.get_Id().toString());
               row.addAttribute("className", doc.getClassName(),
JSONResultSetRow.TYPE_STRING, null, doc.getClassName());
               row.addAttribute("ModifiedBy", doc.get_LastModifier(),
JSONResultSetRow.TYPE_STRING, null, doc.get_LastModifier());
               row.addAttribute("LastModified",
doc.get_DateLastModified().toString(), JSONResultSetRow.TYPE_TIMESTAMP, null,
doc.get_DateLastModified().toString());
               row.addAttribute("Version", doc.get_MajorVersionNumber() + "." +
doc.get_MinorVersionNumber(), JSONResultSetRow.TYPE_STRING, null,
doc.get_MajorVersionNumber() + "." + doc.get_MinorVersionNumber());
               row.addAttribute("{NAME}", doc.get_Name(),
JSONResultSetRow.TYPE_STRING, null, doc.get_Name());
               row.addAttribute("ContentSize", doc.get_ContentSize(),
JSONResultSetRow.TYPE_INTEGER, null, null);

               jsonResults.addRow(row);
            }

         } catch (Exception e) {
            // provide error information
            callbacks.getLogger().logError(this, methodName, request, e);
```

```
            JSONMessage jsonMessage = new JSONMessage(0, e.getMessage(), "This
error may occur if the search string is invalid.", "Ensure the search string is
the correct syntax.", "Check the IBM Content Navigator logs for more details.",
"");
            jsonResults.addErrorMessage(jsonMessage);
         } finally {
            UserContext.get().popSubject();
            callbacks.getLogger().logExit(this, methodName, request);
         }
      }
```

The logic of the custom search service with paging is summarized in Figure 5-17.

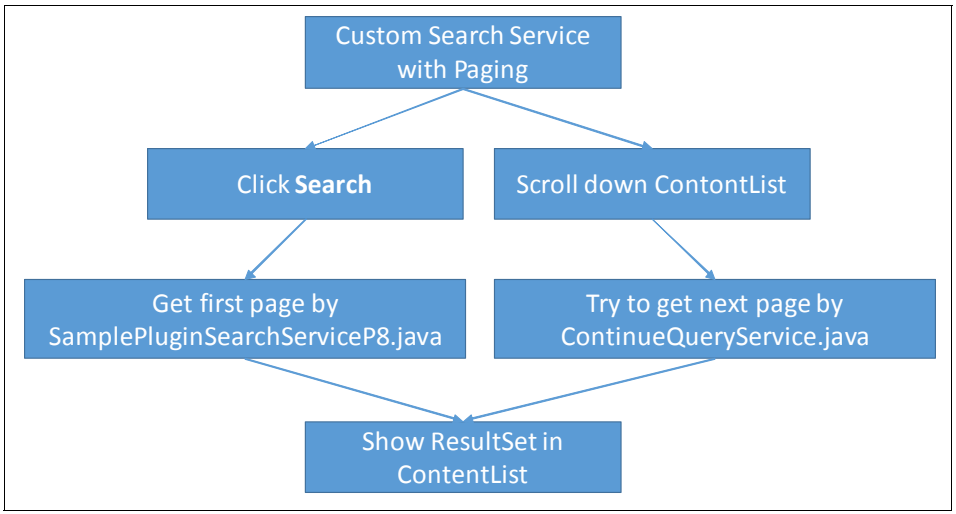



*Figure 5-17 Custom search service with paging logic*

## 5.6.2 Getting the result properties setting from admin configuration

Now the custom search service has the paging function. You can see that all the properties displayed in the ContentList are hard-coded. In the Search tab of IBM Content Navigator, there is a setting for the search result displayed for repositories. See Figure 5-18 on page 208. The displayed properties will be shown in the search result of IBM Content Navigator. This can also be used for the plug-in custom search service if the plug-in code reads and uses it. We enhance the code to do this. Do not select the Class property as it is; it causes an error during the processing of the search result.

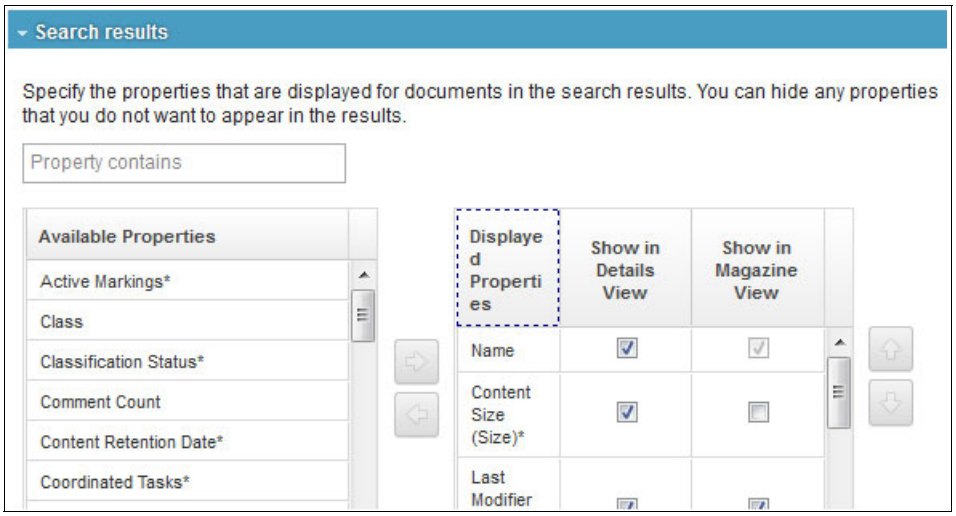



*Figure 5-18 Search result properties display configuration*

We can easily get the search result display configuration information in the plug-in. See Example 5-15.

*Example 5-15 Get search default columns setting in SamplePluginSearchServiceP8.java*

```
RepositoryConfig repoConf = Config.getRepositoryConfig(request);
String[] folderColumns = repoConf.getSearchDefaultColumns();
```

With the folderColumns, we can decide which properties should be searched based on the query string. We can then decide which column will be shown with that setting.

In the sample code, we did not add these properties to the query string. We select the system properties that every document has. See Example 5-16.

*Example 5-16 Build result set columns based on SamplePluginSearchServiceP8.java*

```
      String[] states = new String[1];
      states[0] = JSONResultSetColumn.STATE_LOCKED;

      jsonResultSet.addColumn(new JSONResultSetColumn(" ",
"multiStateIcon", false, states));
      jsonResultSet.addColumn(new JSONResultSetColumn(" ", "17px",
"mimeTypeIcon", null, false));
      //jsonResultSet.addColumn(new
JSONResultSetColumn(resources.getMessage(clientLocale,
"search.results.header.id"), "200px", "ID", null, false));
```

```
      //jsonResultSet.addColumn(new JSONResultSetColumn("Class Name",
"125px", "className", null, false));
      for (String columnString : folderColumns) {
         if (columnString.equals("LastModifier"))
            jsonResultSet.addColumn(new
JSONResultSetColumn(resources.getMessage(clientLocale,
"search.results.header.lastModifiedByUser"), "125px", "ModifiedBy",
null, false));
         else if (columnString.equals("DateLastModified"))
            jsonResultSet.addColumn(new
JSONResultSetColumn(resources.getMessage(clientLocale,
"search.results.header.lastModifiedTimestamp"), "175px",
"LastModified", null, false));
         else if (columnString.equals("MajorVersionNumber"))
            jsonResultSet.addColumn(new
JSONResultSetColumn(resources.getMessage(clientLocale,
"search.results.header.version"), "50px", "Version", null, false));
         else if (columnString.equals("{NAME}"))
            jsonResultSet.addColumn(new JSONResultSetColumn("Name",
"200px", columnString, null, false));
         else {
            jsonResultSet.addColumn(new
JSONResultSetColumn(columnString, "80px", columnString, null, true));
         }
      }
```

After using this, we can show the columns based on configuration (Figure 5-19).

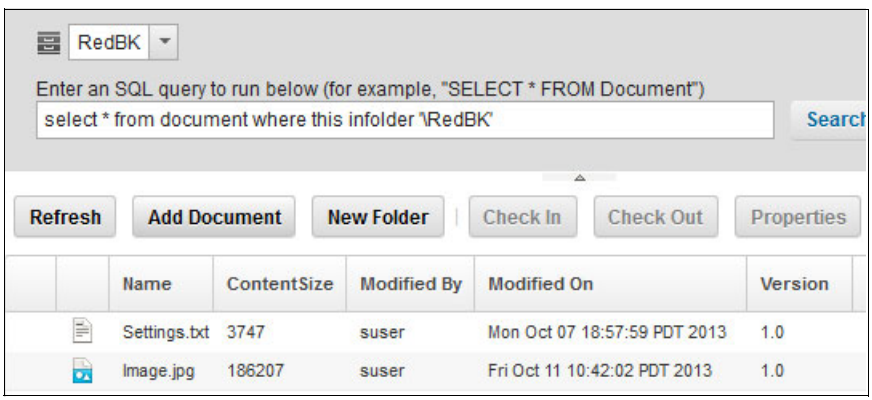



*Figure 5-19 Display search result properties according to configuration*

### 5.6.3 Main files of custom search plug-in

Not all content of the source code in the custom search plug-in are listed in the previous sections. Check them in the project of CustomSearchPlugin associated with this chapter. The following list describes all the main files.

- CustomSearchPlugin.java

  Main class of the custom search plug-in provides all information needed by an IBM Content Navigator plug-in. All extensions and scripts are registered in it.

- SearchService.java

  This service extension allows you to do custom search. The detailed implementations are in `SamplePluginSearchServiceCM.java` and `SamplePluginSearchServiceP8.java`.

- SamplePluginSearchServiceCM.java

  This i copied from the sample plug-in and allows you to search in IBM Content Manager repository with a query string.

- SamplePluginSearchServiceP8.java

  This is copied from the sample plug-in and then enhanced to support a paging search. With it, you can search in IBM FileNet Content Manager repository with query string.

- ContinueQueryService.java

  This request filter extension enables paging search. It filters the request of `/p8/continueQuery`. If pageIterator is saved in session, it returns the next page of the search result.

## 5.7 Conclusion

This chapter describes how to implement a custom search service in IBM Content Navigator plug-in. Sample code is provided for custom search service in the sample plug-in. This chapter describes how to reuse the code in other plug-ins. It also describes how to enhance the code with a paging function for IBM FileNet Content Manager and use the search result properties configuration for repositories.

6

# Creating a feature with search services and widgets

This chapter describes how to create a new feature plug-in including search services and widgets. It continues the topic of Chapter 5, “Building a custom repository search service” on page 173, but focuses on using widgets to build a real scenario. A virtual folder browse pane will be implemented in this chapter.

This chapter covers the following topics:

- Example overview
- Adjusting the layout of the feature
- Creating a tree widget to show a custom tree
- Adding function on the class node to search and show results
- Configuring more modules to the ContentList widget

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# 6.1 Example overview

In Chapter 5, “Building a custom repository search service” on page 173, we created a new feature with search services, imported from the sample plug-in. We did not use the search service in our own page. The CustomSearchPane was created to demonstrate that the search service is working. In this chapter, we enhance the custom search plug-in a real scenario, a *virtual folder browse pane*.

IBM Content Navigator provides a feature called browse pane. Users can change repositories if there are multiple repositories and users can see the folder tree in the left pane. By clicking on a node of the folder tree, the content of that folder will be shown in the ContentList in the right-side pane. Figure 6-1 shows the IBM Content Navigator browse pane.

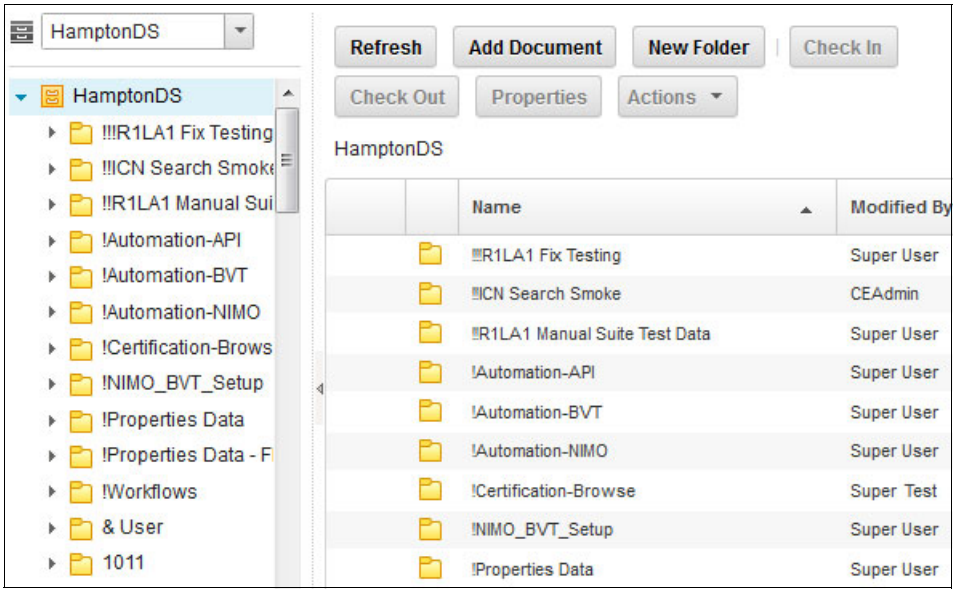



*Figure 6-1 Browse pane of IBM Content Navigator*

With the browse pane, you can navigate through documents and folders in the selected repository. Users can explore the content repository by folder structure.

Sometimes you might want a more flexible method to browse the documents in a repository. In this example, we create an alternative browse functionality called *virtual folders*. There will be a tree, but not based on folders: some nodes are document classes, and some nodes are document property values. Users can build a navigation tree. When users click on a node of the tree, a search is triggered to get the matching documents.

You can leave the tree structure definition to users. They can define their own tree, and then save the tree structure configuration.

In this chapter, we show a basic virtual folders example. The tree structure is hard-coded, but you can enhance it to implement more functions. We enhance the custom search plug-in example to implement this feature.

## 6.2 Adjusting the layout of the feature

For Dojo widgets, an HTML template is defined. The layout of the widget is defined in this template file. The `dijit._TemplatedMixin` helps you to create widgets quickly and easily. IBM Content Navigator uses this to build widgets.

When we created the custom feature pane for a custom search plug-in in Chapter 5, “Building a custom repository search service” on page 173, a template HTML file was created automatically by the Eclipse plug-in. The name of the file is `CustomFeaturePane.html` and the code for the feature is shown in Example 6-1.

*Example 6-1 CustomFeaturePane.html in custom search plug-in*

```
<div class="ecmCenterPane">
   <div data-dojo-type="idx/layout/BorderContainer"
data-dojo-props="gutters:true">
      <div data-dojo-attach-point="searchArea"
data-dojo-type="dijit/layout/ContentPane"
           data-dojo-props="splitter:true,region:'top'"
class="sampleSearchArea">
         <div data-dojo-attach-point="repositorySelectorArea"
class="sampleRepositorySelectorArea"></div>
         <label for="${id}_queryString"
data-dojo-attach-point="cm8HelpText" style="display:none">Enter an
XPath query to run below (for example, &amp;quot;/NOINDEX&amp;quot;)</label>
         <label for="${id}_queryString"
data-dojo-attach-point="p8HelpText" style="display:none">Enter an SQL
query to run below (for example, &amp;quot;SELECT * FROM
Document&amp;quot;)</label>
         <div class="sampleQueryInputArea">
            <input id="${id}_queryString"
data-dojo-attach-point="queryString"
data-dojo-type="dijit/form/TextBox"></input>
           <button data-dojo-attach-point="searchButton"
data-dojo-type="dijit/form/Button"
```

```
                data-dojo-attach-event="onClick: runSearch"
class="searchButton">
              ${messages.search}
           </button>
        </div>
     </div>
     <div data-dojo-attach-point="resultsArea"
data-dojo-type="dijit/layout/ContentPane"
data-dojo-props="region:'center'">
        <div data-dojo-attach-point="searchResults"
data-dojo-type="ecm/widget/listView/ContentList"

data-dojo-props="emptyMessage:'${messages.folder_is_empty}'">
        </div>
     </div>
  </div>
</div>
```

The layout of this template is shown in Figure 6-2.

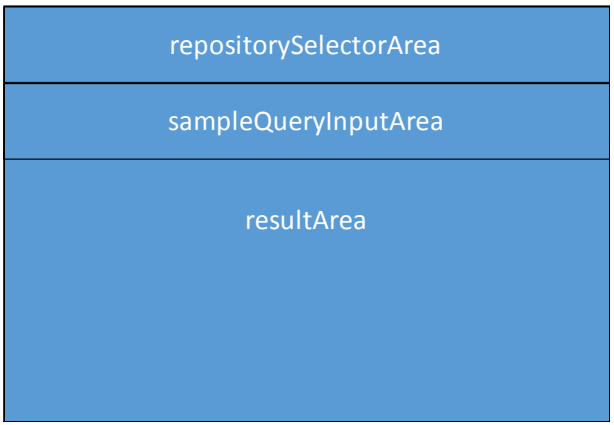



*Figure 6-2 Layout of CustomFeaturePane.html*

The following components are in the template:

- data-dojo-attach-point

  This is the reference to the created Document Object Model (DOM) element. The value can be used in the widget as a property. For example, you can find cm8HelpText in `CustomFeaturePane.js` by using `this.cm8HelpText`.

- data-dojo-type

  This provides a reference to a predefined Dojo widget. For example, the search button uses the existing `dijit/form/Button widget`.

- data-dojo-attach-event

  This provides the mechanism to handle an event of the widget. For example, the onClick event of the search button will be handled by the runSearch function in `CustomFeaturePane.js`.

- data-dojo-props

  This defines properties of the widget.

The layout we use for the new feature pane of virtual folder browsing is similar to the IBM Content Navigator browse pane. Figure 6-3 shows this layout.

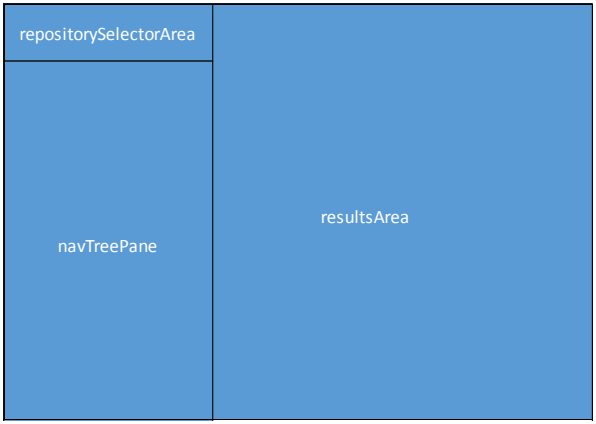



*Figure 6-3 Layout of virtual folder browsing feature*

For this example, the template HTML file is modified as in Example 6-2.

*Example 6-2 Template for virtual folder browse pane*

```
<div class="ecmCenterPane" data-dojo-attach-point="containerNode">
   <div data-dojo-type="idx.layout.BorderContainer"
data-dojo-attach-point="container" class="contentPane" gutters="false"
design="sidebar">
      <div data-dojo-type="dijit.layout.ContentPane" region="leading"
class="paneNoOverflow" splitter="true">
         <div data-dojo-type="dijit.layout.BorderContainer"
gutters="false" class="navContainer">
            <div data-dojo-type="dijit.layout.ContentPane"
data-dojo-props="region:'top'" >
```

```
            <div data-dojo-attach-point="repositorySelectorArea"
class="sampleRepositorySelectorArea"></div>
            </div>
           <div data-dojo-attach-point="searchSelectorArea"
class="navContainerBottomBar" data-dojo-type="dijit.layout.ContentPane"
region="center">
             <div data-dojo-type="dijit.layout.ContentPane"
id="navTreePane" ,  style="width:100%;height:100%;overflow:auto;",
data-dojo-attach-point="navTreePaneObj" ></div>
           </div>
        </div>
      </div>
      <div data-dojo-type="dijit.layout.ContentPane" region="center">
        <div data-dojo-attach-point="navResult"
data-dojo-type="ecm.widget.listView.ContentList" emptyMessage="folder
is empty"></div>
      </div>
   </div>
</div>
```

To accomplish this, we need to create a new feature in the custom search plug-in. The following steps create the feature. The complete dialog is shown in Figure 6-4 on page 217.

1. Make sure the IBM Content Navigator Eclipse plug-in is configured.
2. Right-click the **com.ibm.ecm.extension.customsearch** package in the CustomSearchPlugin project in Eclipse and select **IBM Content Navigator** → **NewFeature**.
3. For the class name, enter `VirtualFolderBrowePane`.
4. Try to find any 32*32 PNG picture file as the new feature icon. Select the **Use new feature image** box and then select your image. You see the preview on the big button. Click **OK**.

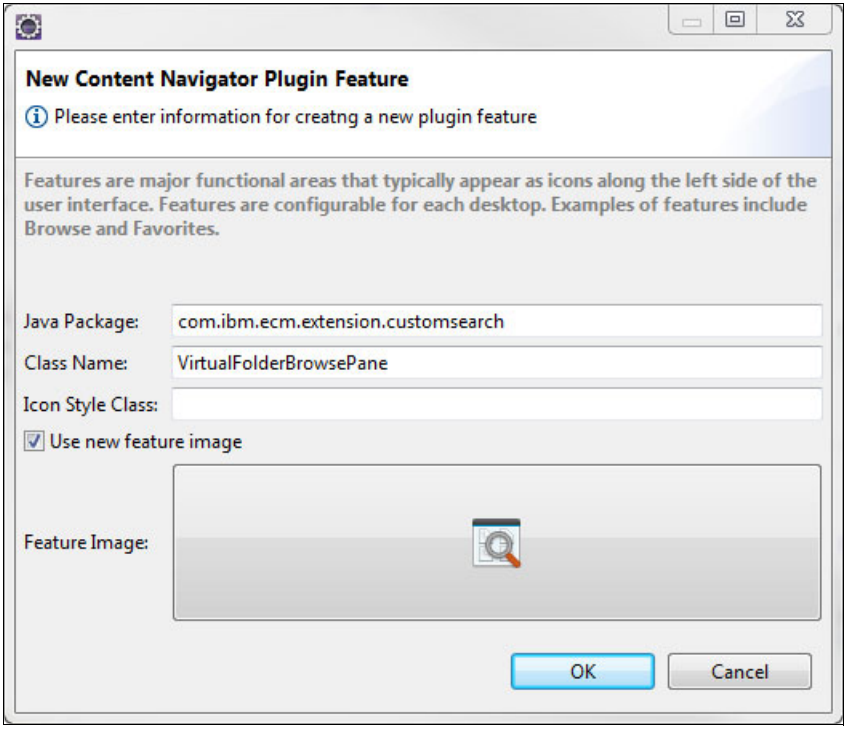



*Figure 6-4 New virtual folder browse pane feature*

5. Complete the wizard; the following files will be created:

```
VirtualFolderBrowsePane.java
VirtualFolderBrowsePane.js
VirtualFolderBrowsePane.html files
```

6. Verify that the getFeatures() method in CustomSearchPlugin.java looks like Example 6-3. If not, modify it manually.

*Example 6-3 getFeatures method of CustomSearchPlugin.java*

```
public com.ibm.ecm.extension.PluginFeature[] getFeatures() {
   return new com.ibm.ecm.extension.PluginFeature[] {new
com.ibm.ecm.extension.customsearch.CustomFeaturePane(), new
com.ibm.ecm.extension.customsearch.VirtualFolderBrowsePane()};
}
```

7. The `VirtualFolderBrowsePane.html` file that is generated by Eclipse plug-in contains no content except an empty `div` element. Replace the content of this file with the code from Example 6-2 on page 215. This will adjust the layout to the design. If you want, you can build the plug-in and try to see if the template layout works and make any needed adjustments.
8. The feature icon file will be copied to the following package:

   `com.ibm.ecm.extension.customsearch.WebContent.images package`

   You can also add icons to that package. Then, the icon definition will be in the `CustomSearchPlugin.css` file as shown in Example 6-4.

   *Example 6-4 CSS definition for icon*

```
.CustomSearchPluginLaunchIcon {
   width: 32px;
   height: 32px;
   background: url('images/search.png') no-repeat;
}
```

9. Define the icon for feature in `VirtualFolderBrowsePane.java` as shown in Example 6-5.

   *Example 6-5 Icon definition for feature*

```
public String getIconUrl() {
   return "CustomSearchPluginLaunchIcon";
}
```

## 6.3 Creating a tree widget to show a custom tree

In 6.2, “Adjusting the layout of the feature” on page 213, we create the needed layout. The repository selector-related code is already in `CustomFeaturePane.js` file. We can reuse it in the `VirtualFoldedrBrowsePane.js` file.

The next step is to create a custom tree. The tree will be created with the Dojo Tree dijit. The Tree dijit needs to have a TreeStore to manage data and nodes. The tree data initialization is a JSON string. It defines the original blank tree data.

There are three properties for each node to be displayed:

- name: Display name
- id: ID for the node
- children: Children nodes data

Example 6-6 shows the JSON that defines the initial tree structure that we use.

*Example 6-6 Original tree data structure*

```
var data ="{\"name\": \"Multiple Demension Tree\",\"id\":
\"root\",\"children\": [{\"name\": \"My Navigator\",\"id\":
\"my_navigator\", \"children\":[]}]}";
```

The tree needs an object store model to manage data and to service the display. We use a memory store here. It is a built-in object store provided by Dojo. The code is shown in Example 6-7.

*Example 6-7 Using memory object store*

```
this.TreeStore = new Memory({
   data: [ json.parse(data) ],
   getChildren: lang.hitch(this,function(object){
   return object.children;
})
});
```

We create a sample tree (Figure 6-5) to show folders and classes combined. With this, users can navigate documents by folders and classes.

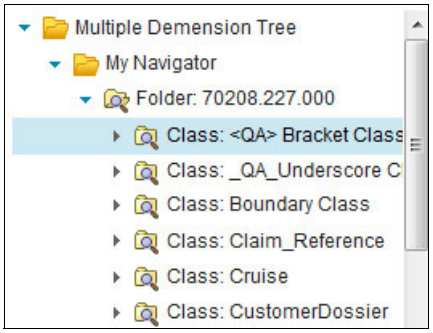



*Figure 6-5 Custom tree with folders and classes*

To build this tree, get the folder list and class list. IBM Content Navigator provides a model layer API that you can use to handle data in the repositories.

To get folders from the root folder requires the root folder ID. The root folder ID can be retrieved by IBM Content Navigator API as shown in Example 6-8.

*Example 6-8 Set rootItemId to root folder.*

```
var rootItemId = this.repository.rootFolderId || "/";
```

With the root folder ID, you can get the top-level folder by calling the IBM Content Navigator API as shown in Example 6-9.

*Example 6-9 Get sub folders of root folder*

```
this.repository.retrieveItem(rootItemId, lang.hitch(this,
function(rootFolder) {
   this.repository.rootFolder=rootFolder;
   rootFolder.retrieveFolderContents(true, callbackGetFolders);
}), null, null, null, this._objectStore ? this._objectStore.id : "");
```

You can use `rootFolder.retrieveFolderContents` to get all its subfolders. With callbackGetFolders, you set the folders into tree nodes. The code is in Example 6-10.

*Example 6-10 callbackGetFolders, set folders into tree nodes.*

```
var callbackGetFolders = lang.hitch(this, function(resultset)
{
   this.mainfolders=[];
   for(row in resultset.items)
   {
      var element= {
            value:resultset.items[row].id,
            label:resultset.items[row].name,
            name:"Folder: "+resultset.items[row].name,
            id:resultset.items[row].id,
            criterionType:"Folder",
            children:[]
      }
      this.mainfolders.push(element);
   }
   this.setupNavTree();
});
```

The code to get the classes is shown in Example 6-11. The callbackGetClasses is a callback function that inserts class data into the tree. The code is shown in Example 6-12 on page 221.

*Example 6-11 Get classes of repository*

```
this.repository.retrieveContentClasses(callbackGetClasses);
```

*Example 6-12 callbackGetClasses*

```
var callbackGetClasses = lang.hitch(this, function(contentClasses)
{
   this.classnames=[];
   for(docclass in contentClasses)
   {
      var element= {
            value:contentClasses[docclass].id,
            label:contentClasses[docclass].name,
            name:"Class: "+contentClasses[docclass].name,
            id:contentClasses[docclass].id,
            criterionType:"Class",
            children:[]
      }
      this.classnames.push(element);
   }
});
```

To show the classes that the customer wants to display in the custom tree, you can add filter logic in this code.

You might notice that the icon of the custom tree is not the same as the IBM Content Navigator folder icon. We show how to use a different icon and also how to create the tree with an object store. Example 6-13 on page 222 shows how to override the getIconClass() method of tree. It returns different icons for different types of nodes. The example also shows how to use the new icon resource. Example 6-14 on page 222 shows how to manually add icon resources into the IBM Content Navigator plug-in project.

**File types:** Not all resource file types are supported by IBM Content Navigator plug-in mechanism. Supported file types for now are `.png`, `.jpg`, `.jpeg`, `.gif`, `.json`, `.js`, `.css`, `.xml`, and `.html`.

*Example 6-13 Create tree with object store and different icons*

```
this.navTree = new Tree({
   model: TreeModel,
   onOpenClick: true,
   persist: false,
   getIconClass: lang.hitch(this, function(item,opened)
      {
         if(item.id != "root" && item.id !="my_navigator")
            return (opened ? "searchFolderOpenIcon" :
archFolderCloseIcon");
         else
            return (opened ? "dijitFolderOpened" :
"dijitFolderClosed");
      })
}, "divTree");
```

In JavaScript code, icon string is a class name in the CSS file. The icon classes definition in the CSS file is shown in Example 6-14. It shows that the icon files are in the images path. They are actually put into the following location:

`com.ibm.ecm.etension.customsearch.WebContent.images`

*Example 6-14 icon definitions in CustomSearchPlugin.css*

```
.searchFolderOpenIcon,
searchFolderOpenIcon {
   background-image: url(images/SearchFolderOpened.png);
   width: 16px;
   height: 16px;
}
.searchFolderCloseIcon,
searchFolderCloseIcon {
   background-image: url(images/SearchFolderClosed.png);
   width: 16px;
   height: 16px;
}
```

The example shows how to build a simple custom tree. Based on this mechanism, you can build your own tree with other data. For example, you can build the tree with values from a property, then users can navigate documents by property values. You can also build the kind of property value tree with hierarchical structure. To add more enhancements, you can save the tree configuration for each user. Then, users can define their own document navigation tree. This way to manage document navigation is flexible.

## 6.4 Adding function on the class node to search and show results

Currently, the tree is displayed when changing the repository selector, but so far no action has been defined when a node is selected in the tree.

In IBM Content Navigator, when users click the node of the folder tree, a search or several searches are triggered. The content of the folder is retrieved and displayed in the ContentList.

For this virtual folder browse feature example, we also want to get functionality that is similar to the IBM Content Navigator browse pane. If you click the node of custom tree, a search is triggered and the results are displayed in the ContentList. The search criteria will be built according to the node properties.

To do this, we need to connect the search method to the custom tree. With Dojo connect, we can bind a DOM event with a function. In Example 6-15, the onClick event of a custom tree node is bounded with the executeSearch() method. The executeSearch() method will build the query string based on node information and then search to get the result. Example 6-15 shows how to connect the tree onClick event to the search function. The executeSearch() method will run the search based on the folder or class that users click.

*Example 6-15 Connect tree onClick event to search function*

```
this.connect(this.navTree, "onClick", lang.hitch(this, function(item) {
   if(item.id != "root" && item.id !="my_navigator")
   {
      this.executeSearch(item);
   }
}));
```

The search uses the existing search service in the custom search plug-in that is built in Chapter 5, “Building a custom repository search service” on page 173. It supports searching with a query string. The service for IBM FileNet Content Manager supports paging and displays the property configuration.

When a user clicks on a folder node, the search passes the folder as a parameter to find all documents in that folder. When a user clicks on a class node, the search passes both the folder and class as the parameters to find documents. Example 6-16 on page 224 shows this logic. After building the parameters, the runSearch() method will be launched to do the search against the repository.

*Example 6-16 Build parameters for search*

```
executeSearch:function(item){
   var node = this.navTree.getNodesByItem(item);
   var path = node[0].tree.path;
   var docClassName="";
   var mainFolderID="";
   for(i=2;path[i]!=undefined;i++)
   {
      if(path[i].criterionType=="Class")
      {
         docClassName=path[i].value;
      }else if(path[i].criterionType == "Folder")
      {
         mainFolderID=path[i].value;
      }
   }
   this.runSearch(docClassName,mainFolderID);
},
```

In the previous sample, we needed to enter the entire query string manually. But in this tree-node-triggered search, the query string needs to be generated by the code. Example 6-17 shows how to build the query string.

*Example 6-17 Build query string*

```
if( this.repository.type == "cm" ){
   var scoperule="/* ";
   var baserulestart='[(@SEMANTICTYPE IN (1))';
   var ruleEnd="]"
   var folderrule='INBOUNDLINK[@LINKTYPE = "DKFolder"]/@SOURCEITEMREF =
';
   var attributerule="";
   if(docClassName!="")
      scoperule='/'+docClassName+" ";
   var query = scoperule+baserulestart;
   if(attributeName!="" && attributeName!=undefined)
   {
      attributerule='((@' +attributeName+" = "+attributeValue +'))'
      query = query +" AND "+attributerule;
   }
   if(mainFolderID!="")
   {
               itemid =mainFolderID.split(" ")[6];
               mainFolderID =itemid.substr(0, itemid.length-2);
      folderrule = folderrule+'""'+mainFolderID+'"';
```

```
      query = query +" AND "+folderrule;
   }
   query +=ruleEnd;
   requestParams.query = query;
}else if(this.repository.type=="p8"){
   var query = "Select * from ";
   if ( docClassName && docClassName.length > 0 ){
      query += docClassName;
   }else{
      query += "DOCUMENT ";
   }
   if( mainFolderID || ( attributeName && attributeValue) ){
      query += " where "
   }
   if( mainFolderID && mainFolderID.length >0 ){
      var folderID = mainFolderID.substr(  mainFolderID.length-38,
mainFolderID.length );
      query += " this INFOLDER " + folderID ;
   }
   requestParams.query=query;
}
```

After that, when a user clicks on a node in the custom tree, the search is launched and results will be shown in the ContentList.

## 6.5 Configuring more modules to the ContentList widget

The ContentList widget is a important widget in IBM Content Navigator. It is based on the Dojo GridX widget but contains many enhancements. There are also some modules through which ContentList can be modified and configured. GridX is a data grid widget to display data. It also supports a module mechanism. It has some modules and you can extend or build your own modules to GridX. ContentList modules are as follows (they are described in Chapter 5, “Building a custom repository search service” on page 173):

- ContentList: This is the core module to show data and launch actions.
- Toolbar: Provide a toolbar containing buttons.
- Bar: Provide the bar capability to arrange ContentList widgets.
- Breadcrumb: Provide the breadcrumb function.
- DocInfo: Show document details.

- FilterData: Provide ability to filter data.
- InlineMessage: Show messages inline.
- TotalCount: Show the total count of the results. It is hidden for repositories that cannot provide the results total count.
- ViewDetail: View module to show detail view.
- ViewMagazine: View module to show magazine view.
- ViewFilmStrip: View module to show filmstrip view.
- RowContextManu: Grid module to display context menu.
- DndFromDesktopAddDoc: Grid module to allow users to “drag-and-drop” from desktop to grid.
- DndRowMoveCopy: Grid module to drag-and-drop a row, then move or copy when dropping the document or documents.
- DndRowCopy: Grid module to drag-and-drop a row, then copy when dropping the document or documents.
- DndDropOnly: Grid module extends DndRowMoveCopy. It can disallow the dragging of rows.

In this section, we configure more modules of the ContentList in the sample code. Example 6-18 shows the ContentList modules. The code is imported from the sample plug-in.

*Example 6-18 ContentList modules*

```
getContentListModules: function() {
   var viewModules = [];
   viewModules.push(ViewDetail);
   viewModules.push(ViewMagazine);
   if (ecm.model.desktop.showViewFilmstrip) {
      viewModules.push(ViewFilmStrip);
   }
   var array = [];
   array.push(DocInfo);
   array.push({
      moduleClass: Bar,
      top: [
         [
            [
               {
                  moduleClass: Toolbar
               },
               {
```

```
                moduleClasses: viewModules,
                "className": "BarViewModules"
             }
          ]
       ]
    ]
 });
 return array;
},
```

The following modules are available:

- ViewDetail
- ViewMagazine
- ViewFilmStrip
- DocInfo
- Toolbar
- Bar
- DndRowMoveCopy
- DndFromDesktopAddDoc
- RowContextMenu

The Bar module allows users to arrange ContentList modules below or above the ContentList grid. In the sample code, there is one line of modules set above the ContentList. The first is ToolBar, the second is view modules for ContentList. The `"className": "BarViewModules"` means giving a class name to view modules. Users can set any class name here. Bar module processes the string to <table> <tr>, and <td> where the arrays determines how many <table>, <tr>, and <td> to create.

We can then add more lines and more modules. There is one more module of FilterData in the first line. The second line is Breadcrumb. The third line is InlineMessage. Then, the TotalCount module is added to the bottom of the ContentList grid. Example 6-19 shows the updated ContentList modules.

*Example 6-19 Add more modules to ContentList*

```
var array = [];
array.push(DocInfo);
array.push({
   moduleClass: Bar,
   top: [
      [
         [
            {
               moduleClass: Toolbar
```

```
            },
            {
               moduleClass: FilterData,
               "className": "BarFilterData"
            },
            {
               moduleClasses: viewModules,
               "className": "BarViewModules"
            }
         ]
      ],
      [
         [
            {
               moduleClass: Breadcrumb,
            }
         ]
      ],
      [
         [
            {
               moduleClass: InlineMessage
            }
         ]
      ]
   ],
   bottom: [
            [
               [
                  {
                     moduleClass: TotalCount
                  }
               ]
            ]
         ]
});
return array;
```

After modifying the code, build and deploy the plug-in. Figure 6-6 shows the updated interface. A filter data component is shown on the top. The inline message shows that 50 items are in the result set of this page. The TotalCount module shows that the quantity of all results is 59. When you scroll down the ContentList, a continuous query will be launched to get the last 9 results.

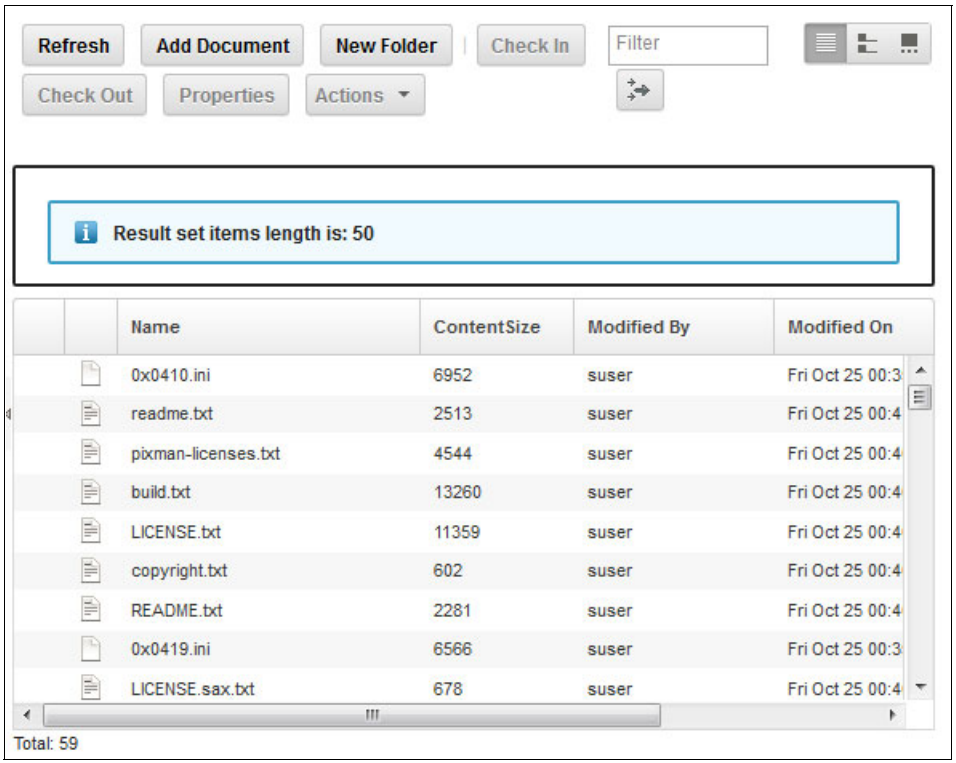



*Figure 6-6 Updated custom search results display interface*

The Breadcrumb module must set parentFolder or searchTemplate in the result set. When you click on the link, the open method of that item is run. If users want to use the breadcrumb function for other conditions, you can write your own module, and add it here.

The InlineMessage module shows messages of warning, information, confirm, and error with the setMessage() method. Example 6-20 shows how many items are in the result set.

*Example 6-20 Using InlineMessageModule to show messages*

```
var inlineMessageModule = this.navResult.getContentListModule(
"inlineMessage" );
if (inlineMessageModule){
   inlineMessageModule.clearMessage();
   inlineMessageModule.setMessage("Result set items length is: "
+resultSet.items.length, "info");
}
```

The TotalCount module shows how many total results are found. Set the totalCount and totalCountType in the result set. Then it will show in the ContentList. IBM FileNet Content Manager V5.2, or later, has an option that can be specified in a query string to return the result size by using the `pageIterator.getTotalCount()` method. The option is similar to `OPTION (COUNT_LIMIT 1000)`.

**Return null:** Be aware that if the result size is less than the page size, the pageIterator.getTotalCount() will return null.

Example 6-21 shows the query with the count limit specified.

*Example 6-21 IBM FileNet Content Manager V5.2 COUNT_LIMIT option*

```
select * from document where this infolder '\Test' OPTIONS (COUNT_LIMIT
1000)
```

Example 6-22 shows the sample code to set data in the result set for the TotalCount module to display. This module can also be used for other data by setting these two values in the result set.

*Example 6-22 Sample for TotalCount module setting*

```
jsonResultSet.put("totalCount", totalCount);
jsonResultSet.put("totalCountType", "total");
```

Figure 6-7 shows the user interface for the completed feature.

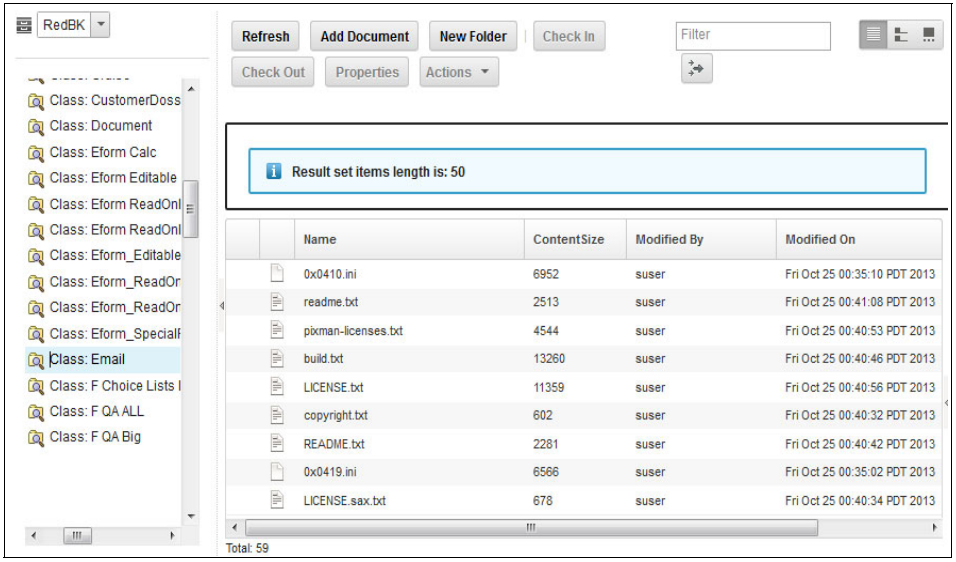



*Figure 6-7 User interface with the completed feature implemented*

## 6.6 Conclusion

In this chapter, we enhance the custom search plug-in to provide a new feature for virtual folder browsing. To do this, we adjust the layout, create a custom tree, connect the search method to the tree node, and build the query string automatically. We also show how to use modules to customize the ContentList widget in code. With the sample code for this chapter, you can build your own custom document navigation plug-in.

The sample code for Chapter 4, “Developing a plug-in with basic extension points” on page 117 shows how to use a folder to manage different type documents. But, this chapter shows that you can manager different type documents by document class also. The users can also navigate documents by document class.

Appendix C, “Core code for the custom search plug-in project” on page 519 provides core code from two files for the custom search plug-in project for your:

- VirtualFolderBrowsePane.js
- Enhanced SamplePluginSearchServiceP8.java code

For complete code, download the additional material associated with this book. See Appendix D, “Additional material” on page 535.

7

# Implementing request and response filters and external data services

This chapter introduces request and response filters. These filters enable you to modify a request before it is sent to the service and modify the response after it is received from the service. In IBM Content Navigator, the external data service (EDS) is a special implementation of the request and response filters. This chapter also describes an EDS implementation. It is important to know when to implement your own request and response filters and when to use EDS. This chapter provides guideline as when to implement one versus the other.

This chapter covers the following topics:

- Request and response filter overview
- External data service (EDS) overview
- Example overview
- Setting up the sample EDS project
- Choice lists
- Property field validation
- Setting field properties
- Ordering properties on the add dialog
- Filter for large choice lists
- Deployment and configuration

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# 7.1 Request and response filter overview

In IBM Content Navigator, when executing a service call, a request is sent to the service and the response is sent back from the service to the client tier. These responses and requests are formatted in JSON. In general, the IBM Content Navigator framework creates the requests and responses in a standard JSON format that is defined by the service call and the framework.

In some instances, you might want to alter the requests and responses to modify the data that is being sent to or returned from the service. You can create a plug-in to filter a request that is made to a service or to filter a response that is received from a service. For example, you might want to check the documents that are included in the request and perform an extra level of security check based on external information. For the response, you might want to implement extra error handling code when the service is not able to perform a specific request. You can also define special formatting for dedicated columns of a result set through modifying the JSON output by using special Dojo classes. A request or response filter stands between the client and the service, and modifies the responses and requests as needed. A filter is always assigned to one service or a list of services within IBM Content Navigator.

Provided services are listed in the `struts-config.xml` file, which is in the following directory:

`<WAS_InstallPath>/profiles/<nameOfProfile>/installedApps/<nameOfCell>/navigator.ear/navigator.war/WEB-INF`

The <action-mappings> section of the `struts-config.xml` file lists actions that can be filtered. The name of the action that must be returned by the getFilteredServices() method is the <action path> tag.

In this chapter, two response filter samples are implemented:

- 7.8, “Ordering properties on the add dialog” on page 260
- 7.9, “Filter for large choice lists” on page 266

## 7.1.1 When to implement request and response filters

Implement request and response filters if you want to do these types of tasks:

- Reorder properties in the dialog.
- Control data in the action that is not included in EDS, such as Teamspace Builder pane.
- Add custom control in the document list view when open a folder in Browse view (for example, set different default sort columns for different folders).

**Note:** Before you proceed with learning about and implementing EDS in the next section, be aware that many implementations should probably be done as request and response filters. You can use the EDS as an example of the request and response filter implementation.

## 7.2 External data service (EDS) overview

The two parts for the EDS are as follows:

- The EDS plug-in that is used to enable the EDS function
- The EDS web application that is used to get external data and set the property behavior

The EDS plug-in uses request and response filters to enable a service to provide the EDS functions. Often, you implement your own request and response filters. However, sometimes, EDS can simplify your implementation.

The following EDS samples are implemented in this chapter:

- 7.5.1, “Simple choice lists” on page 246
- 7.5.2, “Dependant choice lists” on page 255
- 7.6, “Property field validation” on page 256
- 7.7, “Setting field properties” on page 257

### 7.2.1 When to use EDS

Use EDS if you want to perform the following tasks:

- Prefill properties with values.
- Look up the choice list values for a property or dependent property.
- Set minimum and maximum values.
- Set property status, such as read-only, required or hidden.
- Implement property validation and error checking.

In EDS plug-in, it implements many request and response filters. The following request filters are included in EDS plug-in:

- AddItemFilter
- CheckinFilter
- EditAttributesFilter
- SearchFilter
- DispatchItemFilter
- UpdateStepProcessRequestFilter

For each request filter, Table 7-1 describes its service and action.

*Table 7-1 Request filter in EDS plug-in*

| Filter | Service | Action |
|---|---|---|
| AddItemFilter | /p8/addItem<br>/cm/addItem | Add documents and folders |
| CheckinFilter | /p8/checkIn | Checkin documents |
| EditAttributesFilter | /p8/editAttributes<br>/cm/editAttributes | Edit properties |
| SearchFilter | /p8/search<br>/cm/search<br>/od/search | Search |
| DispatchItemFilter | /p8/completeStep | Workflow |
| UpdateStepProcessRequestFilter | /cm/completeStep | Workflow |

In the EDS plug-in, the following response filters are included:

- OpenContentClassFilter
- OpenItemFilter
- OpenProcessorFilter
- UpdateParameterDefsFilter
- UpdateAttributeDefsFilter
- OpenSearchTemplateFilter
- OpenInbasketFilter
- StepProcessVarsFilter
- UpdateStepProcessVarsFilter

Table 7-2 lists the service and action for each response filter.

*Table 7-2 Response filter in EDS plug-in*

| Filter | Service | Action |
|---|---|---|
| OpenContentClassFilter | /cm/openContentClass<br>/p8/openContentClass | Select class |
| OpenItemFilter | /p8/openItem<br>/cm/openItem | Edit properties |
| OpenProcessorFilter | /p8/getLaunchParameters<br>/p8/getStepParameters | Workflow |

| Filter | Service | Action |
|---|---|---|
| UpdateParameterDefsFilter | /p8/getDependentParameters | Dependent parameter in workflow |
| UpdateAttributeDefsFilter | /p8/getDependentAttributeInfo<br>/cm/getDependentAttributeInfo | Dependent properties |
| OpenSearchTemplateFilter | /cm/openSearchTemplate<br>/p8/openSearchTemplate<br>/od/openSearchTemplate | Search |
| OpenInbasketFilter | /p8/getFilterCriteria | Workflow |
| StepProcessVarsFilter | /cm/getStepParameters | Workflow |
| UpdateStepProcessVarsFilter | /cm/getDependentParameters | Dependent parameter in workflow |

According to the implemented EDS request and response filters, the following actions in IBM Content Navigator can be implemented:

- Adding documents and folders
- Checking documents in to the repository
- Editing properties for one or multiple items
- Editing item properties in the viewer
- Using entry templates
- Setting process workflow step properties and workflow filter criteria fields
- Creating or using searches
- Controlling IBM Content Manager OnDemand search folders

**EDS usage:** If you want to do actions other than mentioned here, consider implementing a custom request and response filter.

Several default property behaviors in the EDS plug-in are described in Table 7-3.

*Table 7-3 Default property behaviors in the EDS plug-in*

| Option | Sample | Description |
|---|---|---|
| label | New Label | Label for the property |
| value | New value | Value for the property |
| customValidationError | This field should have the mail format | Description of an invalid reason |
| customInvalidItems | [0,3,4,8] | Invalid multi-value items |
| displayMode | readonly/readwrite | Readonly or readwrite |
| required | true/false | Required or not |
| hidden | true/false | Hidden or not |
| minValue | 1 | Override min value |
| maxValue | 100 | Override max value |
| maxLength | 10 | Override max length |
| format | `\\b[\\w\\.-]+@[\\w\\.-]+\\.\\w{2,4}\\b` | The format in regular expression |
| formatDescription | nicolas@sample.net | Sample of the format |
| choiceList | `[ { "displayName" : "<name>" "active": <true or false> "value" : <value> }, // More choices ... ]` | Choice lists |
| hasDependentProperties | true/false | Has dependent properties |

You can use the property behaviors in your EDS service directly.

## 7.2.2 Sample EDS service provided with IBM Content Navigator

The sample EDS service, which IBM Content Navigator provides, uses the JSON files to store the property behavior definition and data list. In ObjectTypes.json, it includes all the object types that the EDS function will be taken effect. The object types include the class name in the repository, workflow step, IBM Content Manager OnDemand search name, and so on. For every object type, there is a JSON file named with the object name plus `_PropertyData` to store the property behavior definition and data list in this object type. As an example, for the class Document, the JSON file is named `Document_PropertyData.json`.

In the PropertyData JSON file, you can set the behavior for every property. Example 7-1 shows the JSON format in the JSON file.

*Example 7-1 Sample JSON file*

```
[
      {
         "symbolicName" : "<symbolic_name>",
         "value" : <potential new value>,
         "customValidationError" : "Description of an invalid reason",
         "customInvalidItems" : [0,3,4,8], // invalid multi-value items
         "displayMode" : "<readonly/readwrite>",
         "required" : <true or false>,
         "hidden" : <true or false>,
         "maxValue" : <overridden max value>,
         "minValue" : <overridden min value>,
         "maxLength" : <underlying max>,
         "format": <regular expression validating the format>,
         "formatDescription": <human readable description of the
format>,
         "choiceList" :
         {
            "displayName" : "<display_name>",
            "choices" :
            [
               {
                  "displayName" : "<name>"
                  "active": <true or false>
                  "value" : <value>
               },
               // More choices ...
            ]
         }  // Or the special values:
            //
```

```
            //Value  Description
            //-----------------------------------
            //"default"Use class defined choice list (if any).
            //null   Removes the currently assigned choice list.
            //
            // When no choice list is used, Navigator will create a
choice list from the list of valid values, if valid values are defined.
         "hasDependentProperties" : <true or false>,
      },
      // More properties ...
   ]
```

Besides the default property behaviors defined in the EDS plug-in, the sample EDS service provides customized property behaviors. You can also customize the property behavior in your own EDS service. For implementation of the customized property behaviors, see `UpdateObjectTypeServlet.java` in the sample EDS service.

For implementation of the customized property behaviors, see the set up in the `UpdateObjectTypeServlet.java` file in the sample EDS service as follows:

- initialValue: Give an initial value for a property
- validateAs: For now, the sample EDS service implemented a customized property validation NoThrees (see 1.5.1, “Sample external data service” on page 55). This customized property validation does not allow any “3” character in the string, such as `sample3`.
- timestamp: Give a forced timestamp value to the property.
- dependentOn: Specify another property that this property depends.
- dependentValue: Specify the value that this property depends.

## 7.3 Example overview

The insurance company handles property, casualty, and automobile policies. All documents that relate to policies are stored in the insurance company’s Enterprise Content Management (ECM) repository, which uses IBM FileNet Content Manager.

Inbound correspondence is scanned, indexed, and stored in a folder-based filing system by document type, within policy number of a client number.

The filing of inbound documentation is dependant upon the assignment of correct values to the inbound document. An invalid or misspelled word means that the

document will not be filed correctly. Incorrectly filed documents do not enter the appropriate business process, which then leads to delays and additional costs.

The insurance company has decided to implement external data services to manage the consistency of data that is entered during the post-scanning indexing phase of inbound document capture.

The insurance company wants to implement a range of features in its application:

- Choice lists
  - Simple choice lists: For selecting data such as marital status
  - Dependent choice lists: For constraining values such as address data
  - Filter choice lists: For filtering selections in the choice list
- Field validation
  - For validating key data such as policy number and client ID against the main policy administration system
- Reorder the properties in the Add dialog

The implementation of the EDS will reduce the data entry effort and improve accuracy of the document-indexing phase, minimizing the risk of misfiled documents, and reducing the overall cost of the operation.

For the simple choice lists, dependent choice lists, and field validation features, the company can easily implement them with EDS. For the filter choice lists and reorder of properties in the add dialog features, they cannot be implemented with EDS. The company needs to use the response filter plug-in to implement the features.

The company wants to leverage the functionality that is provided by EDS to manipulate the company's property input fields. Several use cases will be fulfilled with EDS capabilities:

- Provide a choice list for all the estimators for a car insurance claim. This choice list can be a list of people from a database table.
  See 7.5.1, "Simple choice lists" on page 246 for details.
- Provide a dependent choice list of categories that belong to one main category. This choice list is to limit the available categories under each main category. Categories can be regions; subcategories are cities with offices.
  See 7.5.2, "Dependant choice lists" on page 255 for details.

- Validate, by format, the claim numbers that are entered; for example, validate whether they contain the exact number and types of characters.
  See 7.6, “Property field validation” on page 256 for details.
- Hide the input property field, which accepts custom text describing the reason a claim is opened, if none of the provided reasons in the choice list make sense.
  See 7.7, “Setting field properties” on page 257 for details.
- Limit the amount of adjusted loss for a newly created insurance claim to a certain value.
  See 7.7.2, “Setting field minimum and maximum values” on page 259 for details.

The following features will be realized by implementing the request and response filter:

- Reorder properties in add dialog. The original order for the properties in the add dialog is not appropriate. It will set a specific order for a specific class.
  See 7.8, “Ordering properties on the add dialog” on page 260 for details.
- Filter choice lists. For the large choice list, it provides filter function for the specific property that has a large choice list.
  See 7.9, “Filter for large choice lists” on page 266 for details.

## 7.4 Setting up the sample EDS project

In this section, we create a new implementation of a EDS based on the sample EDS implementation that is provided with IBM Content Navigator. The sample EDS implementation is in the following location:

`<IBM Content Navigator install>/samples/sampleEDSService`

The provided sample can be imported as a WAR file into the workspace of Eclipse or Rational Application Developer, which is being used throughout this chapter.

The following steps describe the import and setup of the sample project:

1. Click **File** → **Import**, select **Existing project into workspace** as the option, and then click **Next.**
2. In the next window, enter the path to the provided sampleEDSService directory. See Figure 7-1 on page 243.

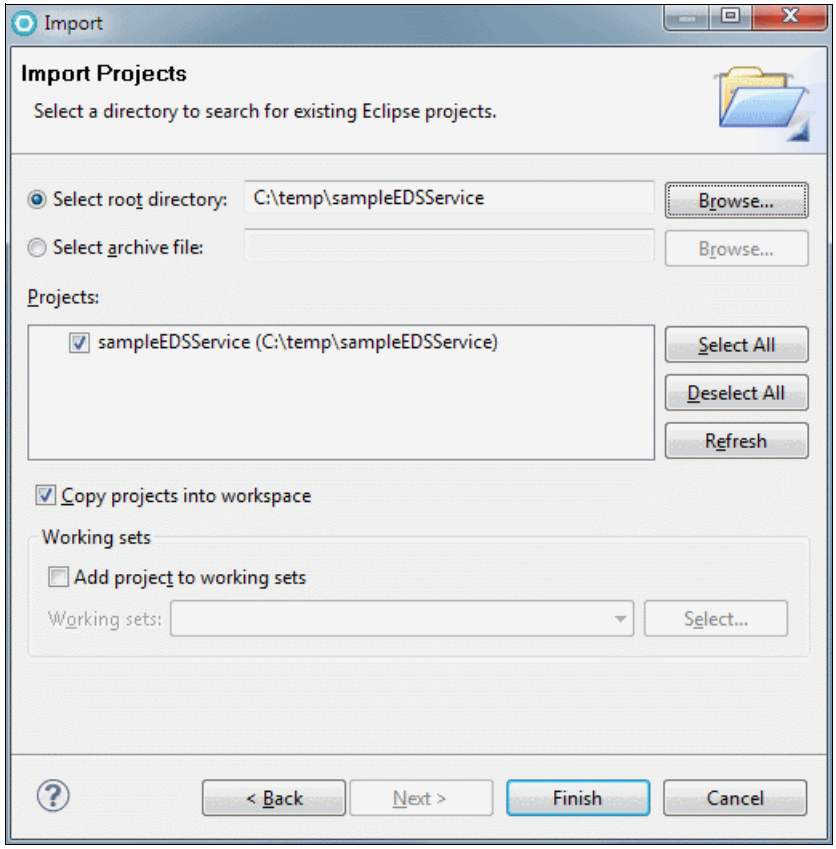



*Figure 7-1 Import project dialog*

3. Click **Finish**. The imported project is shown in Figure 7-2.

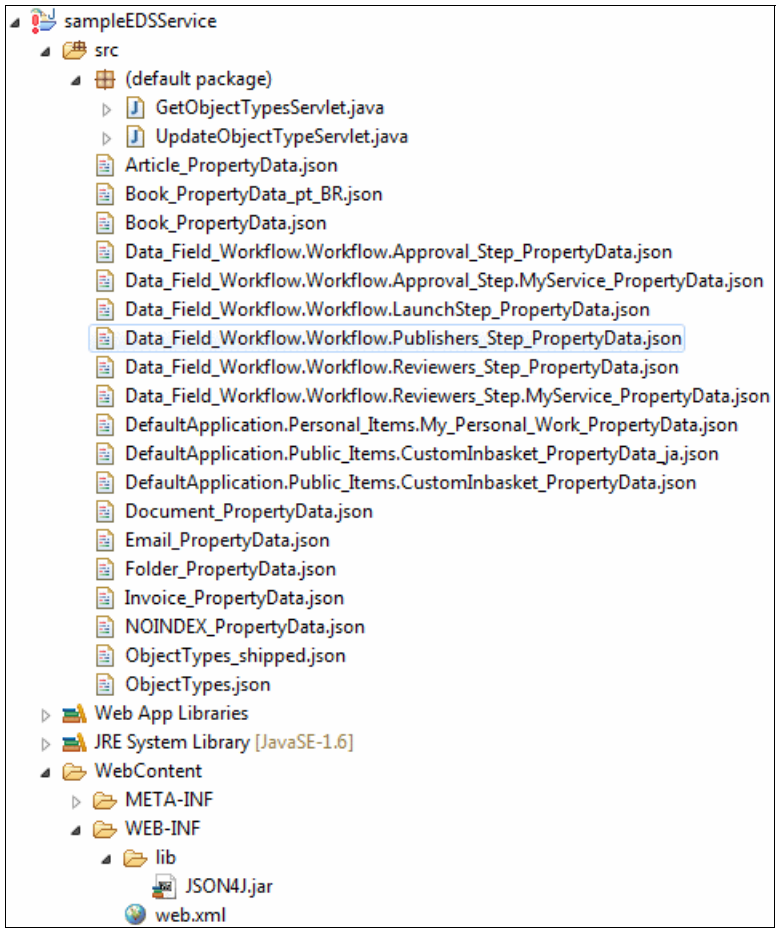



*Figure 7-2 Sample EDS project structure*

4. After importing the existing project, a build path problem occurs with the configured reference to the `j2ee.jar` file, which can easily be fixed. Open the project properties, select **Java Build Path** from the left navigation on the left, and then select the **Libraries** tab where you delete the existing reference to the missing `j2ee.jar` file.
5. Select **Add Library** → **Server Runtime** → **Next** → **WebSphere Application Server v7.0** → **Finish** to add the built-in WebSphere run time that is included with Rational Application Developer to your build path.
6. Rename the project to customEDSService by selecting the project, and then selecting **Refactor** → **Rename** from the top action bar.

If you plan to use Eclipse or another IDE, specify the path to a valid `j2ee.jar` file. The build path should be similar to what is shown in Figure 7-3.

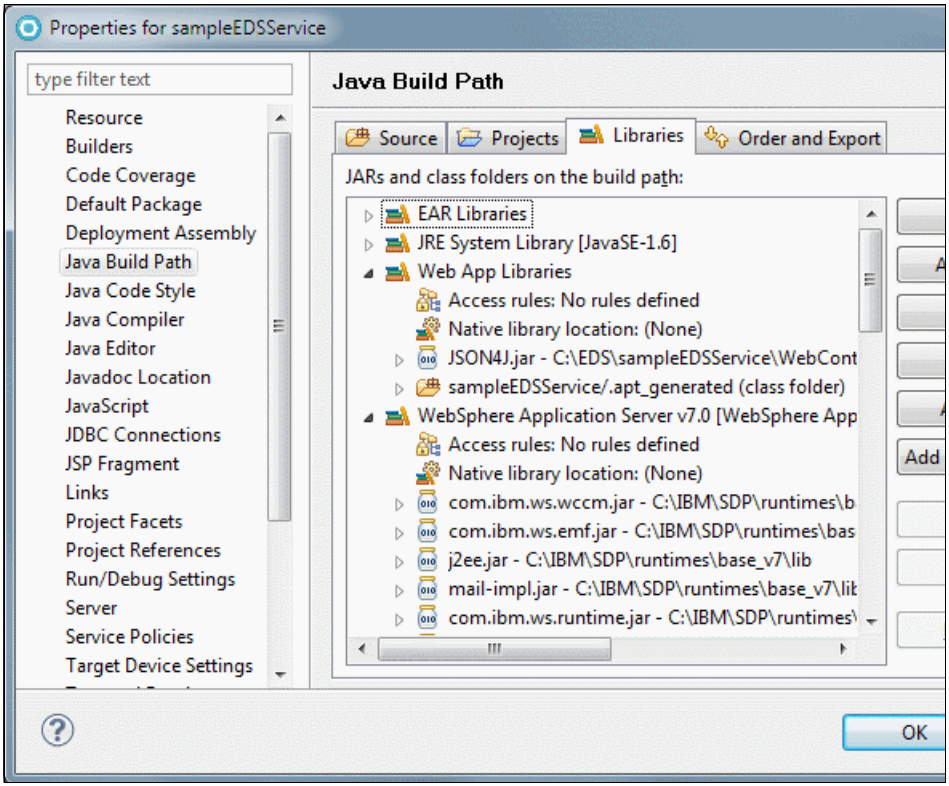



*Figure 7-3 EDS project build path*

Two servlet classes build the entire functionality of the EDS implementation:

- GetObjectTypesServlet: Provide the HTTP GET service to determine the list of object types that are supported by this EDS implementation.
- UpdateObjectTypeServlet: Provide the HTTP POST service that is used to obtain external data and dynamically change, validate, and enrich metadata.

The remaining files in the sample are JSON files that hold the data returned by this EDS implementation, which we are not using except for the `ObjectTypes.json` file. In 7.5, “Choice lists” on page 246, we modify the UpdateObjectTypeServlet class and keep the GetObjectTypesServlet unchanged.

For the insurance company in the sample, we create a data model that consists of two document classes, Claim and Policy. Because we use only the Claim

document class in the sample, this entry is the only one we need to be put to the existing `ObjectTypes.json` file, replacing all the existing entries to be recognized as EDS-supported object type. The new `ObjectTypes.json` file is shown in Example 7-2.

*Example 7-2 ObjectTypes.json*

```
[
   {"symbolicName": "Claim"}
]
```

## 7.5 Choice lists

One of the main features of EDS is to provide values for choice lists. The choice lists can be hierarchical and can also be specified as dependant. This means that the values of one choice list are dependant on the selection of a value for another property.

### 7.5.1 Simple choice lists

The first requirement of the insurance company in our example is to have a list of estimators that can be assigned for a car insurance claim. The list is retrieved from a database table that contains the information about the people. It is required for preparing the web application to be able to connect to the database, by using a data source definition that is assigned to the application through a resource reference. Because the Internet has many samples of how to configure JDBC database access, this information is not covered in this chapter. Instead, we rely on a working data source configuration on the application server that we are deploying to.

The first lines of the UpdateObjectTypeServlet remain unchanged when they load the request JSON and extract some values using JSNON4J. Example 7-3 shows the first lines that are used to load the request and initialize the database connection.

*Example 7-3 Code snippet from sample EDS servlet*

```
protected void doPost(HttpServletRequest request, HttpServletResponse
response)
      throws ServletException, IOException{
   String objectType = request.getPathInfo().substring(1);

   // Get the request json
   InputStream requestInputStream = request.getInputStream();
```

```
   JSONObject jsonRequest = JSONObject.parse(requestInputStream);
   String requestMode = jsonRequest.get("requestMode").toString();
   JSONArray requestProperties =
      (JSONArray)jsonRequest.get("properties");
   JSONArray responseProperties = new JSONArray();
   JSONArray propertyData =
      getPropertyData(objectType, request.getLocale());
```

To read the properties for a specific object type, we implement the getPropertyData() method of the sample to access the database instead of the JSON files. This method is implemented as shown in Example 7-4.

*Example 7-4 Custom getPropertyData reading from database tables*

```
// database constants
private final String schema = "NAVIGATOR";
private final String edsTable = "EDS";
private final String edsChoicesTable = "EDS_CHOICES";

/**
 * Get property data from EDS database tables, replacing the original
 * getPropertyData reading from JSON files.
 *
 * @param objectType
 * @param locale
 * @return The EDS data in sample JSON format
 */
private JSONArray getPropertyData(String objectType, Locale locale) {
   JSONArray jsonPropertyData = new JSONArray();
   Connection con = null;

   try {
      Context ctx = new InitialContext();
      DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/EDSDB");
      con = ds.getConnection();

      // Query the database for EDS data
      Statement stmt = con.createStatement();
      ResultSet results = stmt.executeQuery("SELECT " + edsTable
            + ".OBJECTTYPE," + edsTable + ".PROPERTY," + edsTable
            + ".DISPMODE," + edsTable + ".REQUIRED," + edsTable
            + ".HIDDEN," + edsTable + ".MAXVAL," + edsTable
            + ".MINVAL," + edsTable + ".MAXLEN," + edsTable
            + ".FORMAT," + edsTable + ".FORMATDESC," + edsTable
            + ".HASDEPENDANT," + edsChoicesTable + ".LISTDISPNAME,"
            + edsChoicesTable + ".DISPNAME," + edsChoicesTable
            + ".VALUE," + edsChoicesTable + ".DEPON," + edsChoicesTable
```

```
      + ".DEPVALUE" + " FROM " + schema + '.' + edsTable + ' '
      + edsTable + " LEFT JOIN " + schema + '.' + edsChoicesTable
      + ' ' + edsChoicesTable + " ON " + edsTable
      + ".OBJECTTYPE=" + edsChoicesTable + ".OBJECTTYPE"
      + " AND " + edsTable + ".PROPERTY=" + edsChoicesTable
      + ".PROPERTY" + " AND " + edsChoicesTable + ".LANG='"
      + locale + "'" + " where " + edsTable + ".OBJECTTYPE='"
      + objectType + "'" + " ORDER BY " + edsChoicesTable
      + ".DEPON," + edsChoicesTable + ".DEPVALUE");

String property = null;
String listDispName = null;
boolean firstLoop = true;
String dependentOn = null;
String dependentValue = null;

JSONObject propertyJson = new JSONObject();
JSONObject choiceList = new JSONObject();
JSONArray choices = new JSONArray();

// iterate through the EDS data and build the corresponding JSON
while (results.next()) {
   String propertyTemp = results.getString("property");

   if (firstLoop) {
      property = propertyTemp;
      listDispName = results.getString("listdispname");
      propertyJson = fillBasicProperties(results, property);
      firstLoop = false;
   }

   // check if the property is different to the one
   // in the previous loop
   if (!propertyTemp.equals(property)) {
      if (!choices.isEmpty()) {
         choiceList.put("displayName", listDispName);
         choiceList.put("choices", choices);
         propertyJson.put("choiceList", choiceList);
         listDispName = results.getString("listdispname");
         choiceList = new JSONObject();
         choices = new JSONArray();
      }
      jsonPropertyData.add(propertyJson);
      property = propertyTemp;
      propertyJson = fillBasicProperties(results, property);
   }
```

```
String listDispNameTemp = results.getString("listdispname");
if (!results.wasNull()) {
   if (!listDispNameTemp.equals(listDispName)) {
      choiceList.put("displayName", listDispName);
      choiceList.put("choices", choices);
      propertyJson.put("choiceList", choiceList);

      // close property, add to array and create new
      jsonPropertyData.add(propertyJson);

      listDispName = listDispNameTemp;
      choiceList = new JSONObject();
      choices = new JSONArray();
   }
} else {
   // no choice list attached, continue to next loop
   continue;
}

String dependentOnTemp = results.getString("depon");
String dependentValueTemp = results.getString("depvalue");

// check if there is a dependenOn/Value
// set for the choice list
if ((null != dependentOnTemp && null != dependentValueTemp
      && !dependentOnTemp.isEmpty() && !dependentValueTemp
      .isEmpty())) {
   // historic values are null, set dependentOn/dependentValue
   if (null == dependentOn && null == dependentValue) {
      dependentOn = dependentOnTemp;
      dependentValue = dependentValueTemp;
      propertyJson.put("dependentOn", dependentOn);
      propertyJson.put("dependentValue", dependentValue);
   } else {
      // historic values are not null but different,
      // start new dependent list
      if (!dependentOnTemp.equals(dependentOn)
            || !dependentValueTemp.equals(dependentValue)) {
         dependentOn = dependentOnTemp;
         dependentValue = dependentValueTemp;
         propertyJson = fillBasicProperties(results,
               property);
         propertyJson.put("dependentOn", dependentOn);
         propertyJson.put("dependentValue", dependentValue);
      }
   }
}
```

```
            JSONObject choice = new JSONObject();
            choice.put("displayName", results.getString("dispname"));
            choice.put("value", results.getString("value"));
            choices.add(choice);
         }
         // add the last property
         jsonPropertyData.add(propertyJson);

         stmt.close();
         con.close();

      } catch (NamingException e) {
         e.printStackTrace();
      } catch (SQLException se) {
         System.out.println("SQL Exception:");

         // Loop through the SQL Exceptions
         while (se != null) {
            System.out.println("State  : " + se.getSQLState());
            System.out.println("Message: " + se.getMessage());
            System.out.println("Error  : " + se.getErrorCode());

            se = se.getNextException();
         }
      } finally {
         try {
            if (null != con && !con.isClosed())
               con.close();
         } catch (SQLException se) {
            System.out.println("SQL Exception:");

            // Loop through the SQL Exceptions
            while (se != null) {
               System.out.println("State  : " + se.getSQLState());
               System.out.println("Message: " + se.getMessage());
               System.out.println("Error  : " + se.getErrorCode());

               se = se.getNextException();
            }
         }
      }

      return jsonPropertyData;
   }

   /**
    * read properties from EDS table and start new property
```

```
 *
 * @param results
 * @param property
 * @return
 * @throws SQLException
 */
private JSONObject fillBasicProperties(ResultSet results, String property)
      throws SQLException {
   JSONObject propertyJson = new JSONObject();
   propertyJson.put("symbolicName", property);

   // check all the optional settings
   String displayMode = results.getString("dispmode");
   if (!results.wasNull())
      propertyJson.put("displayMode", displayMode);
   String required = results.getString("required");
   if (!results.wasNull() && '1' == required.charAt(0))
      propertyJson.put("required", true);
   String hidden = results.getString("hidden");
   if (!results.wasNull() && '1' == hidden.charAt(0))
      propertyJson.put("hidden", true);
   Double maxVal = results.getDouble("maxval");
   if (!results.wasNull())
      propertyJson.put("maxValue", maxVal);
   Double minVal = results.getDouble("minval");
   if (!results.wasNull())
      propertyJson.put("minValue", minVal);
   Integer maxLen = results.getInt("maxlen");
   if (!results.wasNull())
      propertyJson.put("maxLength", maxLen);
   String format = results.getString("format");
   if (!results.wasNull())
      propertyJson.put("format", format);
   String formatdesc = results.getString("formatdesc");
   if (!results.wasNull())
      propertyJson.put("formatDescription", formatdesc);
   String hasDependant = results.getString("hasdependant");
   if (!results.wasNull() && '1' == hasDependant.charAt(0))
      propertyJson.put("hasDependentProperties", true);

   return propertyJson;
}
```

In the database are two tables that contain the information necessary for each of the document type properties. Consider the provided database structure as a sample concerning nullable, varchar sizes, and other database column specifics.

The two tables are defined as shown in Figure 7-4 and Figure 7-5.

| Name | Data type | Length | Nullable |
|---|---|---|---|
| OBJECTTYPE | VARCHAR | 64 | No |
| PROPERTY | VARCHAR | 64 | No |
| DISPMODE | VARCHAR | 10 | Yes |
| REQUIRED | CHARACTER | 1 | Yes |
| HIDDEN | CHARACTER | 1 | Yes |
| MAXVAL | DOUBLE | 8 | Yes |
| MINVAL | DOUBLE | 8 | Yes |
| MAXLEN | INTEGER | 4 | Yes |
| FORMAT | VARCHAR | 64 | Yes |
| FORMATDESC | VARCHAR | 128 | Yes |
| HASDEPENDANT | CHARACTER | 1 | Yes |

*Figure 7-4 EDS object properties table definition*

| Name | Data type | Length | Nullable |
|---|---|---|---|
| OBJECTTYPE | VARCHAR | 64 | No |
| PROPERTY | VARCHAR | 64 | No |
| LISTDISPNAME | VARCHAR | 64 | No |
| LANG | CHARACTER | 5 | No |
| DISPNAME | VARCHAR | 128 | No |
| VALUE | VARCHAR | 128 | No |
| DEPON | VARCHAR | 64 | Yes |
| DEPVALUE | VARCHAR | 64 | Yes |

*Figure 7-5 EDS choices table definition*

After the getPropertyData() method is rewritten, the data can be inserted in the tables according to the use case.

In the first sample, insert data in the object properties table and choice-list table. For the object property table, insert one data entry for the Claim object type and the Estimator property. See Figure 7-6.

| OBJECTTYPE | PROPERTY |
|---|---|
| Claim | Estimator |

*Figure 7-6 Simple property definition*

For the choice-list table, insert as many estimator choices as required. This choice list of the estimators appears (using EDS) when a claim processing requires a user to select an estimator for the claim. Figure 7-7 shows the four data entries for the estimator choices.

| OBJECTTYPE | PROPERTY | LISTDISPNAME | LANG | DISPNAME | VALUE |
|---|---|---|---|---|---|
| Claim | Estimator | Estimators | en_US | Estimator1 | Estimator1 |
| Claim | Estimator | Estimators | en_US | Estimator2 | Estimator2 |
| Claim | Estimator | Estimators | en_US | Estimator3 | Estimator3 |
| Claim | Estimator | Estimators | en_US | Estimator4 | Estimator4 |

*Figure 7-7 Simple choice list data*

Now, deploy and test the implemented service. For the deployment, follow the instructions in 7.8, “Ordering properties on the add dialog” on page 260.

Validating the EDS implementation is done by using IBM Content Navigator and calling the add document action in some of the windows. In our test, we use the browse pane to call this action. From the folder structure, we select an existing claim document and want to assign an estimator. During the properties dialog loads, the program calls EDS to ask for modifications to the choice list and other changes that influence the user interface and that are defined for this specific document class.

Figure 7-8 shows the resulting choice list, filled with the values coming from the database served by EDS.

**Important:** Do not implement choice lists in the Content Platform Engine definitions for the same properties that are associated with the choice lists that you provide by using the external data services.

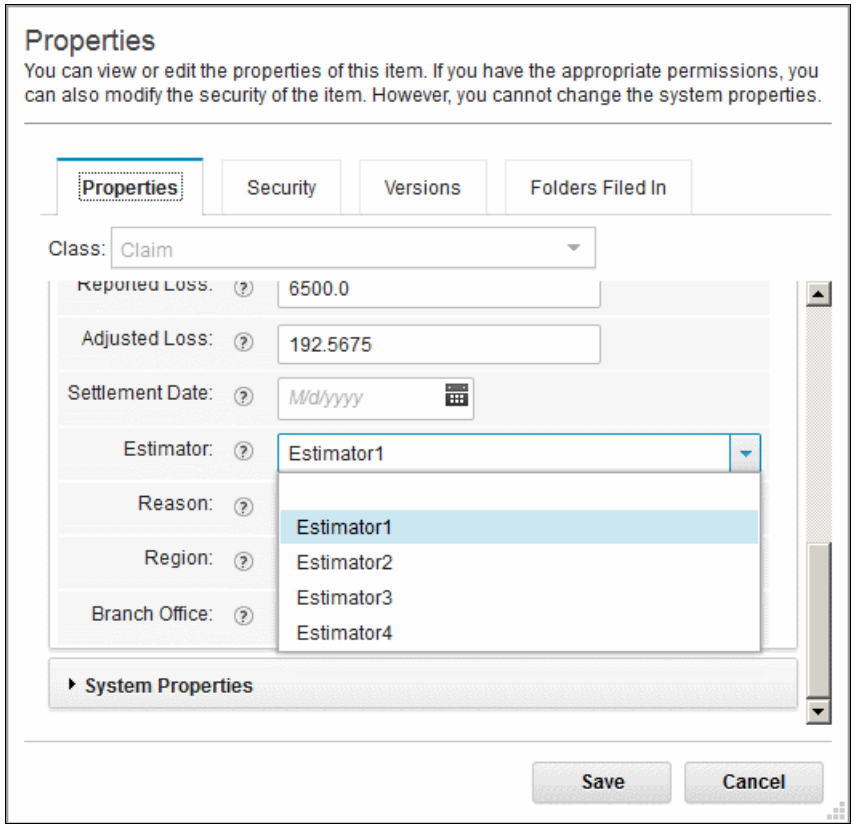



*Figure 7-8 Simple choice list*

## 7.5.2 Dependant choice lists

For dependant choice lists, we expand the first sample and make use of the DEPON and DEPVALUE fields of the choices table, and also the HASDEPENDANT field of the object properties table. Those items indicate that the specific property has a dependancy to another property and its selected value. In this case, we want to show a dependancy between the Region and Branch Office properties and therefore must make inserts in both of our database tables. The inserted data is shown in Figure 7-9 and Figure 7-10.

| OBJECTTYPE | PROPERTY | HASDEPENDANT |
|---|---|---|
| Claim | BranchOffice | |
| Claim | Region | 1 |

*Figure 7-9 Property definition with dependancy*

| OBJECTTYPE | PROPERTY | LISTDISPNAME | LANG | DISPNAME | VALUE | DEPON | DEPVALUE |
|---|---|---|---|---|---|---|---|
| Claim | Region | US Regions | en_US | Midwest | Midwest | | |
| Claim | Region | US Regions | en_US | Western | Western | | |
| Claim | Region | US Regions | en_US | Southwest | Southwest | | |
| Claim | Region | US Regions | en_US | Northeast | Northeast | | |
| Claim | BranchOffice | Western Branch Offices | en_US | Los Angeles | Los Angeles | Region | Western |
| Claim | BranchOffice | Western Branch Offices | en_US | San Francisco | San Francisco | Region | Western |
| Claim | BranchOffice | Western Branch Offices | en_US | Salt Lake City | Salt Lake City | Region | Western |
| Claim | BranchOffice | Western Branch Offices | en_US | Seattle | Seattle | Region | Western |
| Claim | BranchOffice | Western Branch Offices | en_US | Anchorage | Anchorage | Region | Western |
| Claim | BranchOffice | Southwest Branch Offices | en_US | Las Vegas | Las Vegas | Region | Southwest |
| Claim | BranchOffice | Southwest Branch Offices | en_US | Dallas | Dallas | Region | Southwest |
| Claim | BranchOffice | Southwest Branch Offices | en_US | Houston | Houston | Region | Southwest |
| Claim | BranchOffice | Southwest Branch Offices | en_US | Phoenix | Phoenix | Region | Southwest |

*Figure 7-10 Dependent choice list entries*

To test this new data that is entered in the database tables, we call the properties dialog again; the Region property now has a choice list associated to it. See Figure 7-11. As specified in the first table data, the Region property has the hasDependentProperties flag enabled; it triggers IBM Content Navigator to recall EDS each time such a property value changes.

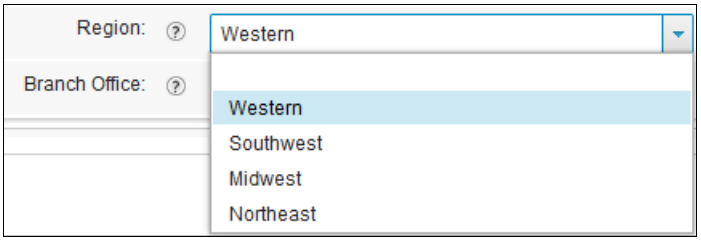



*Figure 7-11 Dependant choice list: Region*

If you now change the value of Region to Southwest, the choice list for Branch Office is updated and reflects the values from the database table.

The choice list shown in Figure 7-12 is the result.

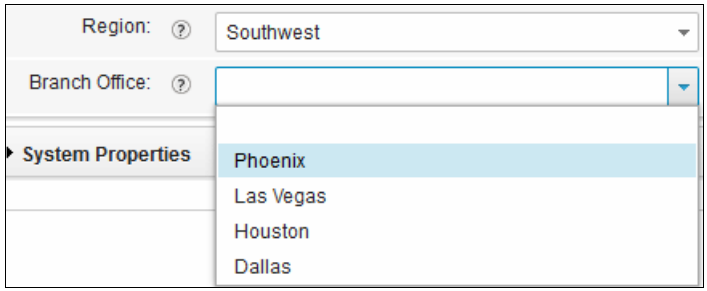



*Figure 7-12 Dependant choice list: Branch Office*

## 7.6 Property field validation

A powerful function of EDS is the ability to implement custom validation mechanisms for property fields, and responding the user with adequate error messages.

In this sample, we want to have the Claim Number property match a specific format that is two numbers, followed by a dash, and then followed by five numbers, for example 01-12345. Therefore, we need to insert data to the EDS object properties table, as in Figure 7-13.

| OBJECTTYPE | PROPERTY | FORMAT | FORMATDESC |
|---|---|---|---|
| Claim | ClaimNumber | \d\d-\d\d\d\d\d | nn-nnnnn |

*Figure 7-13 Validation data*

This step can be tested by using the properties dialog of an existing document of the Claim class. We expect the Claim Number property input field to have a validation error if we enter a string that does not match our defined rules. The syntax of the rule is a regular expression, so you can also specify more complex validation rules. The format description value is available in the tooltip to provide you with a “hint” of what the entered value looks like.

To test the validation, we enter `01-123456` or `01-1234A` as the value for Claim Number. The resulting error is shown in Figure 7-14.

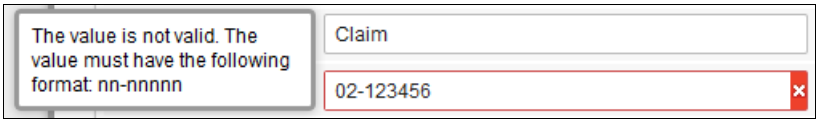



*Figure 7-14 Validation error with tooltip*

# 7.7 Setting field properties

With EDS, you can manipulate certain characteristics of a property field to influence the appearance of your input mask, search view, or on whatever window you manipulate object properties.

## 7.7.1 Setting field status

In the IBM Content Navigator user interface, several input fields have certain properties that influence the appearance. Among those properties, the required, hidden, or read-only status of an input field can be set through EDS. In this sample, we want to dynamically show the Other Reason property if, for the Reason property, the value of Other is selected from the choice list. Therefore we insert data to both the EDS tables, as shown in Figure 7-15 and Figure 7-16.

| OBJECTTYPE | PROPERTY | HASDEPENDANT |
|---|---|---|
| Claim | Reason | 1 |

*Figure 7-15 Property definition: Reason*

| OBJECTTYPE | PROPERTY | LISTDISPNAME | LANG | DISPNAME | VALUE |
|---|---|---|---|---|---|
| Claim | Reason | Reasons | en_US | Accident | Accident |
| Claim | Reason | Reasons | en_US | Vandalism | Vandalism |
| Claim | Reason | Reasons | en_US | Theft | Theft |
| Claim | Reason | Reasons | en_US | Other | Other |

*Figure 7-16 Choice list data with trigger value*

When you reopen the properties dialog for a claim document, the newly configured choice list is displayed with the Other option. See Figure 7-17.

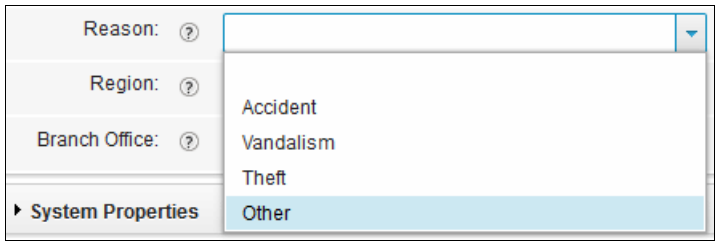



*Figure 7-17 Hidden property choice list*

We want to use the Other option as the trigger to display the Other Reason property input field. To make the specific value be a trigger for another property, we add simple code functionality as a sample. The updated code section is inserted in the original doPost() method of the UpdateObjectServlet class. The code must be executed for initialNewObject, initialExistingObject, and inProgressChnages, and therefore is placed inside this original if statement that limits the request mode. Example 7-5 shows the code.

*Example 7-5 Code insertion to dynamically hide property field*

```
// Check if property 'Reason' is contained and depending on the
// value set hidden property of 'OtherReason' to true or false
if (overridePropertyName.equals("Reason")) {
   for (int j = 0; j < requestProperties.size(); j++) {
      JSONObject requestProperty =
         (JSONObject) requestProperties.get(j);
      String requestPropertyName = requestProperty.get(
            "symbolicName").toString();

      if (requestPropertyName.equals("OtherReason")) {
         if (overrideProperty.get("value").equals("Other")) {
            requestProperty.put("hidden", false);
         } else {
            requestProperty.put("hidden", true);
         }
         responseProperties.add(requestProperty);
      }
   }
}
```

The sample EDS implementation must be redeployed to incorporate the changes that are made to the Java source code.

The result can be tested by opening the properties dialog again with the Reason value set to Other; you then see that the input field for Other Reason property is no longer hidden. If you select another value for the Reason field, the EDS is called again and switches the property to a hidden state again. The result is shown in Figure 7-18.

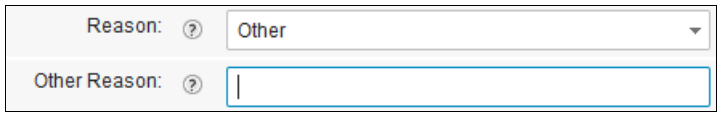



*Figure 7-18 Initially hidden property shows up*

## 7.7.2 Setting field minimum and maximum values

In addition to the input field properties and validation options that are shown in the previous samples, a special range setting is available for integer, float, and date-time properties. The minimum and maximum value for a property field can be set through EDS, where appropriate.

In this sample, we want to restrict the maximum amount that can be set for adjusted loss property value.

To have the maximum value in place, insert one line to the object properties EDS table, as Figure 7-19 shows.

| OBJECTTYPE | PROPERTY | MINVAL | MAXVAL |
|---|---|---|---|
| Claim | AdjustedLoss | 0 | 12,000 |

*Figure 7-19 Maximum value property*

We test the functionality and expect similar results to the other validation sample, which is that a validation error occurs. The associated tooltip gives a hint about what is causing the validation to fail. In our case, we can see the validation fail if we enter an amount that is above 12000. This amount is what we entered to the object properties table as the maximum value for the Adjusted Loss property. The resulting validation error is shown in Figure 7-20.

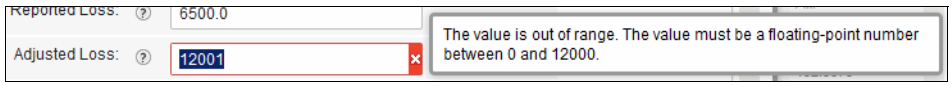



*Figure 7-20 Maximum value validation error*

The configured minimum and maximum value also are visible when you hover the mouse over the question mark icon next to the input field caption. This

hovering displays a similar tooltip, which provides information about the data type and the minimum and maximum values.

**Note:** If a minimum or maximum value is specified for the property in the repository, the service cannot make the setting less restrictive. That is, the service can set the minimum or maximum only to a larger or smaller value. It cannot decrease or increase the minimum or maximum value.

## 7.8 Ordering properties on the add dialog

In this sample, we work with the CustomerDossier class. The original order of the properties is shown in Figure 7-21 where the Company property is under the CustomerNumber property. This sample changes the property order by moving the Company property above the CustomerNumber property.

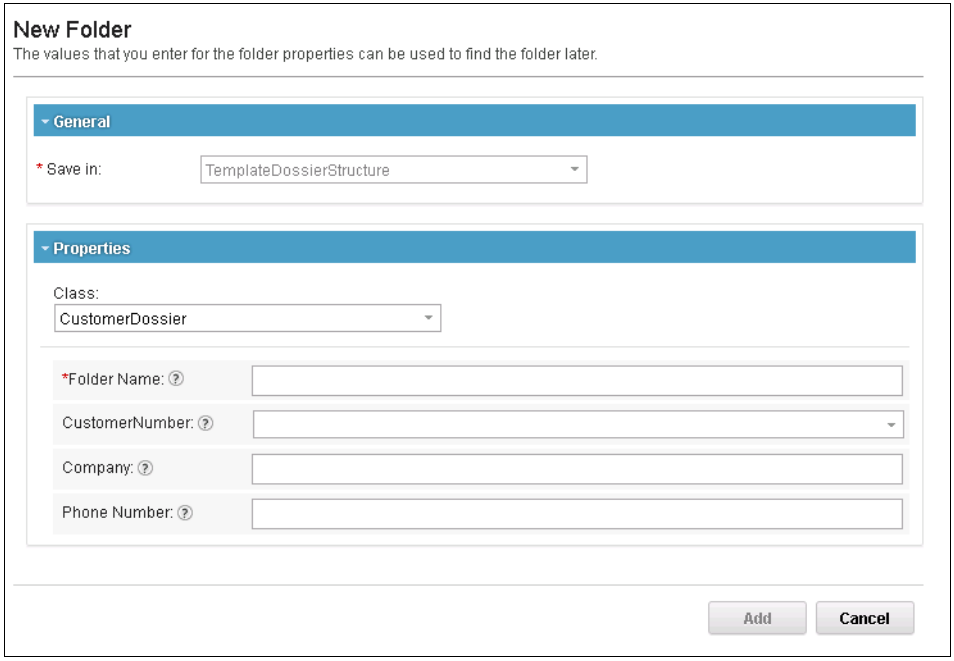



*Figure 7-21 CustomDossier class original order of properties*

EDS can control only the property data and behavior; it cannot change the order of properties in the dialog. To implement this function, a response filter plug-in is needed.

**Note:** Before you try to implement changes with EDS, first determine what types of tasks EDS supports. See 7.2.1, “When to use EDS” on page 235. If EDS does not support a type of task, implement your own request and response filters instead.

After the modified plug-in is applied, the order of the properties will be as shown as Figure 7-22.

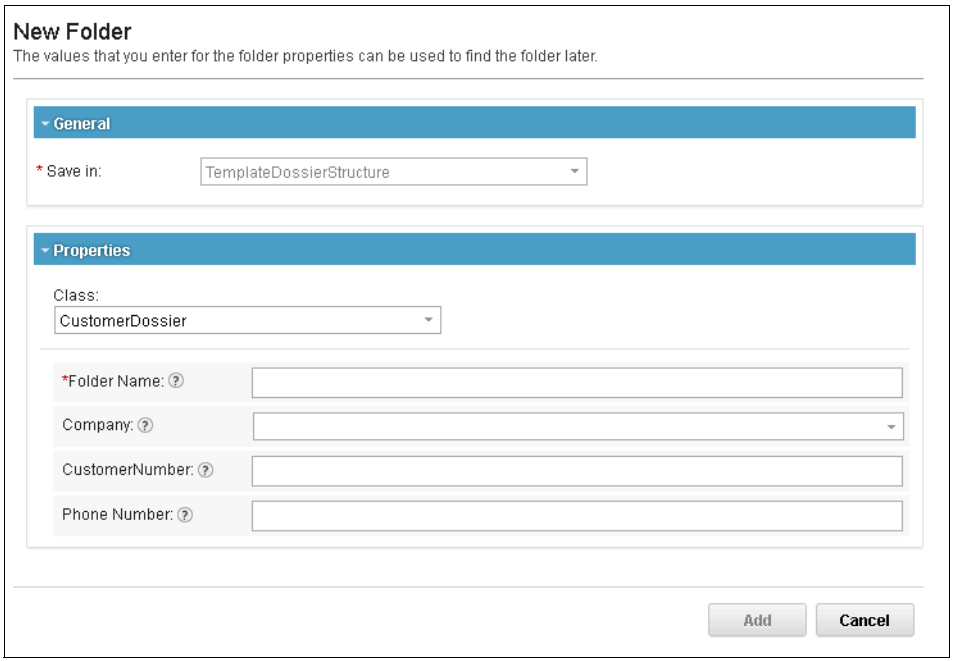



*Figure 7-22 Updated properties order for CustomDossier*

To implement this response filter plug-in, use the following steps:

1. Create a new IBM Content Navigator Plug-in project.

   For the sample plug-in, enter `SampleFilterPlugin` as the Descriptive Name. Enter the remaining information for the plug-in project as shown in Figure 7-23.

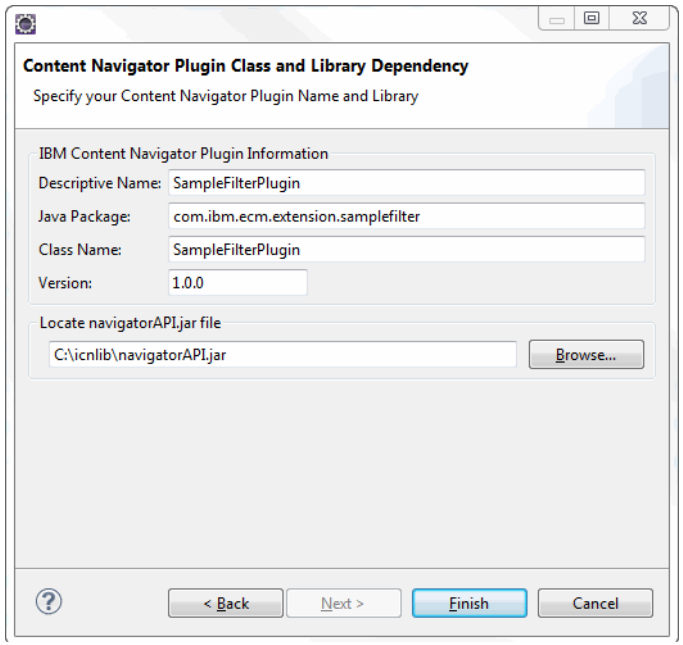



*Figure 7-23 Create a new plug-in project*

2. Create a new response filter in the newly created project by selecting **IBM Content navigator** → **Server Extensions** → **New Response Filter**. See Figure 7-24 on page 263.

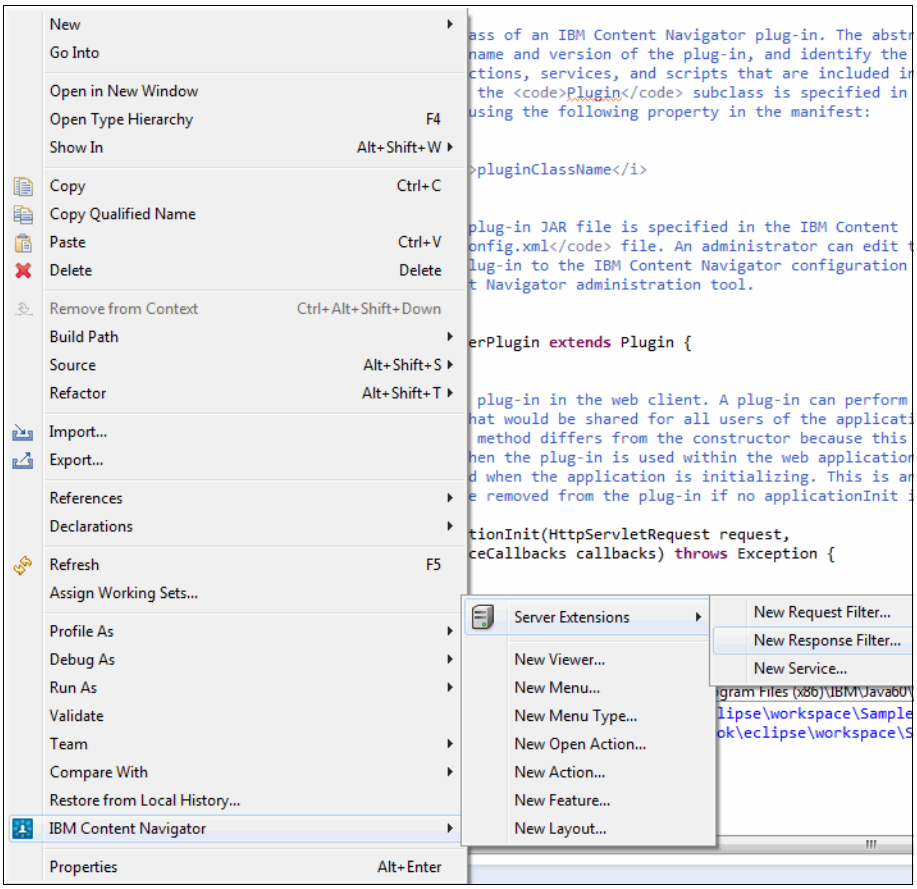



*Figure 7-24 Create a new response filter*

3. In the Response Filter setting dialog, enter the Java Package name and Class Name. Select the services that are extended by this filter.

   For this sample, enter the information as shown in Figure 7-25 on page 151. The following two services are selected to get class information from the repository:

   ```
   /cm/openContentClass
   /p8/openContentClass
   ```

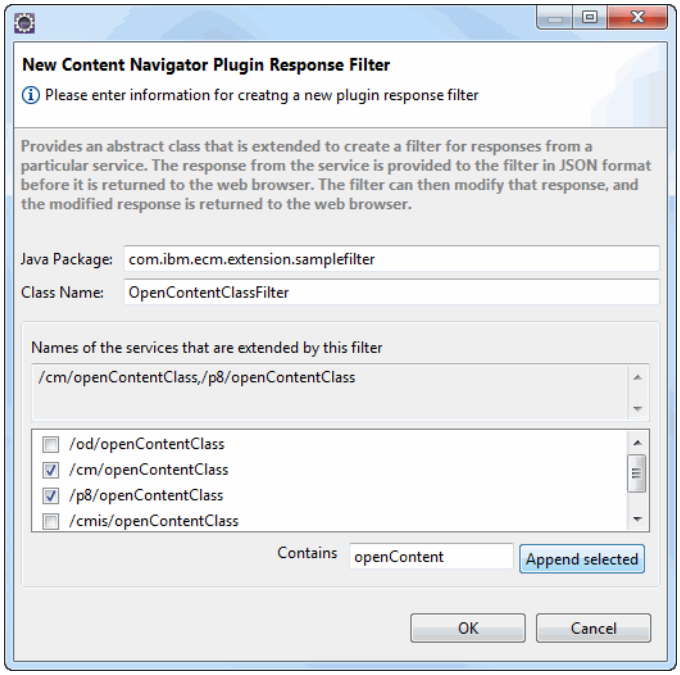



*Figure 7-25 Set up the new response filter*

4. The `OpenContentClassFilter.java` file is created. In this Java file, you need to implement the changing of the properties order that is stored in the jsonResponse parameter. All the property information is stored in criterias property in jsonResponse. The position of the property in the array determines the order that shows in the dialog. This sample sets the specified position for the properties in the array to realize the reorder function in the add dialog.

Example 7-6 shows the sample code for this plug-in. To illustrate how it works and make the sample simple, the class name and properties order is hard-coded. You can make it more configurable.

*Example 7-6 Response filter code snippet to change property display order*

```
package com.ibm.ecm.extension.samplefilter;

import javax.servlet.http.HttpServletRequest;

import com.ibm.ecm.extension.PluginResponseFilter;
import com.ibm.ecm.extension.PluginServiceCallbacks;
import com.ibm.json.java.JSONObject;
```

```
import com.ibm.json.java.JSONArray;

/**
 * Provides an abstract class that is extended to create a filter for
 * responses from a particular service. The response from the service
 * is provided to the filter in JSON format before it is returned to
 * the web browser. The filter can then modify that response, and the
 * modified response is returned to the web browser.
 */
public class OpenContentClassFilter extends PluginResponseFilter {

   /**
    * Returns an array of the services that are extended by this
    * filter.
    */
   public String[] getFilteredServices() {
      return new String[] {
"/cm/openContentClass","/p8/openContentClass" };
   }

   public void filter(String serverType, PluginServiceCallbacks
callbacks,
         HttpServletRequest request, JSONObject jsonResponse) throws
Exception {

      // fill in the initial property values to pass to EDS.  These
      // would be the default values

      JSONArray jsonProperties =
         (JSONArray)jsonResponse.get("criterias");
      String symbolicName =
         jsonResponse.get("template_name").toString();

      if (symbolicName.equals("CustomerDossier")) {
         updateJSONPropertiesOrder(jsonProperties);
      }

   }

   public void updateJSONPropertiesOrder(JSONArray jsonProperties){
      String[] PropertiesOrder = {"Company", "CustomerNumber",
      "PhoneNumber"} ;
      for (int i = 0; i < PropertiesOrder.length; i++) {
         for (int j = 0; j < jsonProperties.size(); j++) {
            JSONObject jsonProperty =
```

```
            (JSONObject)jsonProperties.get(j);
           if
(jsonProperty.get("name").toString().equals(PropertiesOrder[i])) {
              JSONObject jsonEDSProperty =
                 (JSONObject)jsonProperties.get(j);
              jsonProperties.add(i+1, jsonEDSProperty);
              jsonProperties.remove(j+1);
           }
        }

     }
  }
}
```

## 7.9 Filter for large choice lists

Figure 7-26 shows the drop-down input combo box used for the choice list of a single value property.

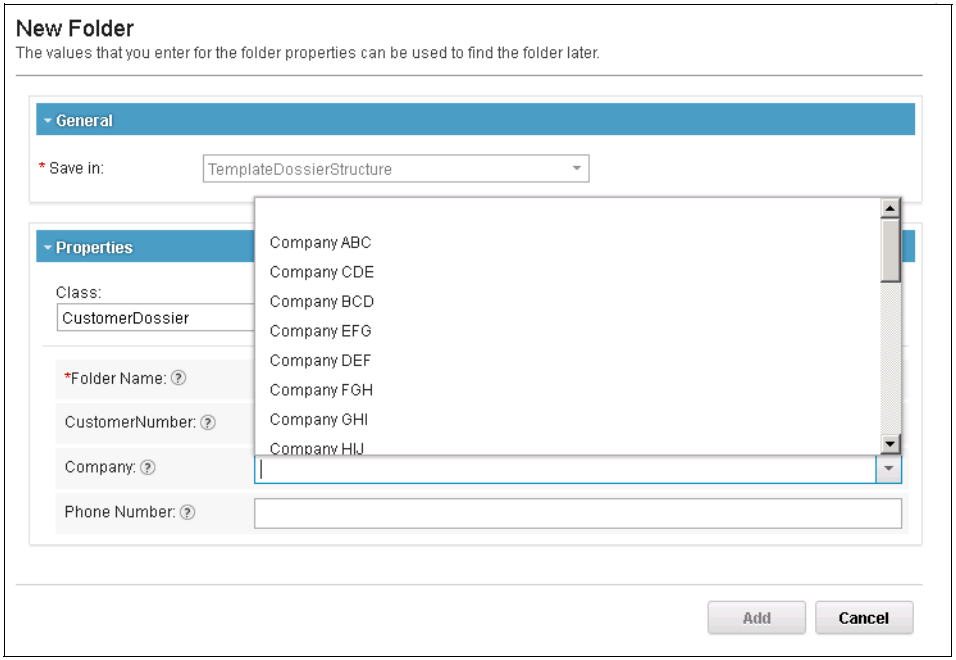



*Figure 7-26 Choice list with normal behavior*

The drop-down input combo box can provide the “start with” filter for the choice list. For a large choice list, if you want the “contain” filter for the choice list, you need a plug-in to change the behavior. To not break the existing drop-down input combo box, in this sample, we provide a new filter widget for the specified property. After the plug-in is registered, the choice list for the Company property will be similar to Figure 7-27.

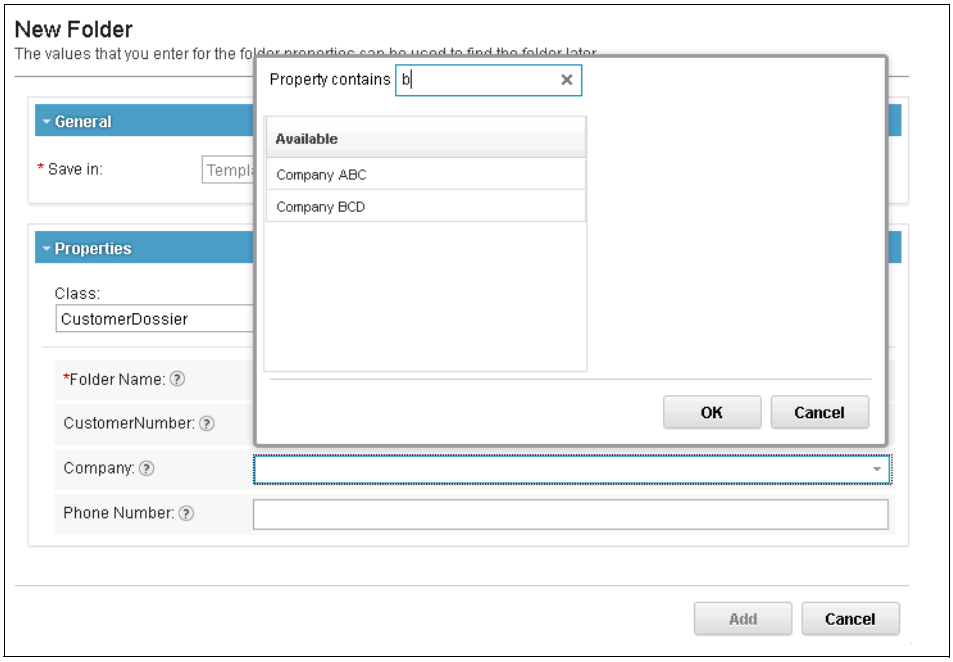



*Figure 7-27 Modified choice list displaying only properties matching restrictions*

In this plug-in sample, there is a response filter to specify the class and property that the filter plug-in works on. The plug-in also provides the customized widgets for the filter choice dialog. To invoke the customized widgets, the plug-in overrides one `_createSearchChoiceListEditor()` method in the `SinglePropertyEditorFactory.js` file.

To implement this response filter plug-in, complete the following steps:

1. Create a new IBM Content Navigator Plug-in project, SampleSearchChoiceListPlugin with the project information as shown in Figure 7-28.

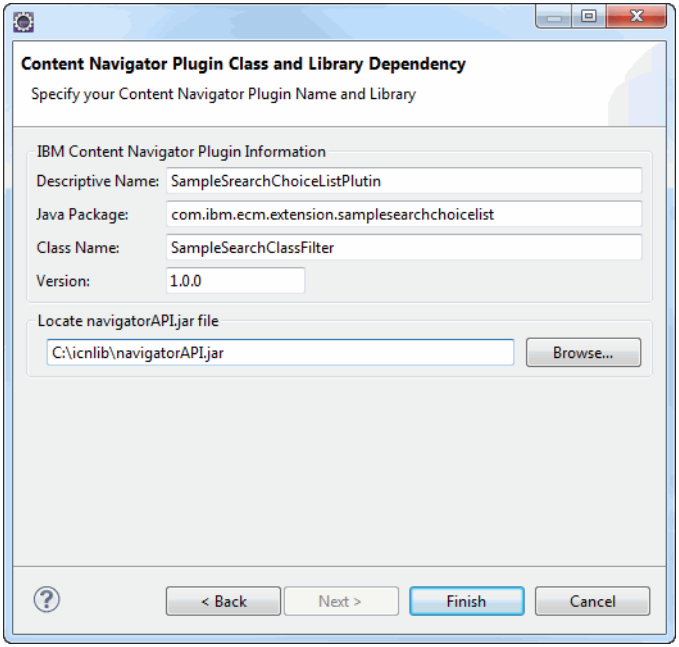



*Figure 7-28 SampleSearchChoiceListPlugin project information*

2. Create a new response filter in the project. In the Response Filter setting dialog (shown in Figure 7-29 on page 269), do these tasks:
   - Enter the Java Package name and Class Name.
   - Select the services that are extended by this filter. In this sample, the following two services are selected to get class information from the repository:

     ```
     /cm/openContentClass
     /p8/openContentClass
     ```

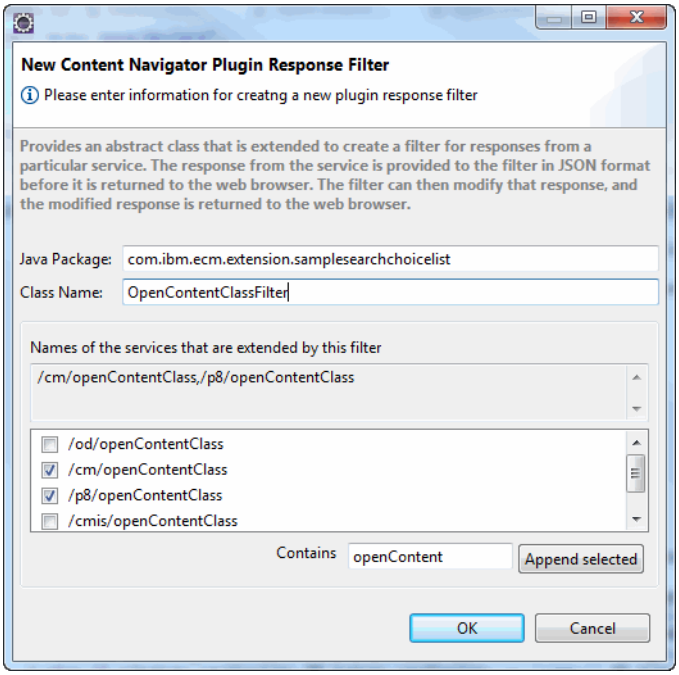



*Figure 7-29 Response filter for SampleSearchChoiceListPlugin project*

A Java file named `OpenContentClassFilter.java` is created. In this sample, the response filter is used to specify the class that the plug-in will apply to. It also adds a **"**`cardinality: SEARCHCHOICE`**"** option to the property in the response JSON object. Example 7-7, shows the code snippet for this plug-in. To illustrate how it works and make the sample simple, the class name and properties are hard-coded. You can make it more configurable. Some comments have been removed for a shorter code snippet.

*Example 7-7 Response filter for a choice list items based on "contains" restriction*

```
package com.ibm.ecm.extension.samplesearchchoicelist;

import javax.servlet.http.HttpServletRequest;

import com.ibm.ecm.extension.PluginResponseFilter;
import com.ibm.ecm.extension.PluginServiceCallbacks;
import com.ibm.json.java.JSONArray;
import com.ibm.json.java.JSONObject;

public class OpenContentClassFilter extends PluginResponseFilter {

   /**
```

```
    * Returns an array of the services that are extended by this
    * filter.
    */
   public String[] getFilteredServices() {
      return new String[] {
"/cm/openContentClass","/p8/openContentClass" };
   }

   public void filter(String serverType, PluginServiceCallbacks
callbacks,
         HttpServletRequest request, JSONObject jsonResponse) throws
Exception {

      // fill in the initial property values to pass to EDS.  These
      // would be the default values
      JSONArray jsonProperties =
         (JSONArray)jsonResponse.get("criterias");
      String symbolicName =
         jsonResponse.get("template_name").toString();

      if (symbolicName.equals("CustomerDossier")) {
         updateJSONFilterOption(jsonProperties);

      }
   }

   public void updateJSONFilterOption(JSONArray jsonProperties){

         for (int i = 0; i < jsonProperties.size(); i++) {
            JSONObject jsonProperty =
             (JSONObject)jsonProperties.get(i);
            if (jsonProperty.get("name").toString().equals("Company"))
{
               JSONObject jsonEDSProperty =
                (JSONObject)jsonProperties.get(i);
               jsonEDSProperty.put("cardinality", "SEARCHCHOICE");
            }
         }

   }
}
```

To invoke the customized widgets, the plug-in overrides one method, which is `_createSearchChoiceListEditor()`. This method is overridden in `SampleSearchChoiceListPlugin.js`. Example 7-8 shows the overridden method.

*Example 7-8 Overridden method _createSearchChoiceListEditor() code snippet*

```
/SampleSearchChoiceListPlugin/src/com/ibm/ecm/extension/samplesearchcho
icelist/WebContent/require(["dojo/_base/declare",
         "dojo/_base/lang",
         "ecm/widget/SinglePropertyEditorFactory",
         "sampleSearchChoiceListPluginDojo/SearchChoicePane"
         ],
function(declare, lang, SinglePropertyEditorFactory, SearchChoicePane)
{
   /**
    * Use this function to add any global JavaScript methods your
plug-in requires.
    */
   SinglePropertyEditorFactory.prototype.createSinglePropertyEditor =
function(kwArgs) {
      var methodName = "createSinglePropertyEditor";
      this.logEntry(methodName, "name: " + kwArgs.name + " label: " +
kwArgs.label + " type: " + kwArgs.dataType + " required: " +
kwArgs.required);
      if (!kwArgs.minMaxValues) {
         kwArgs.minMaxValues = this.getMinMax(kwArgs.minValue,
kwArgs.maxValue, kwArgs.dataType, kwArgs.dataFormat); // call this
before getPromptText
      }
      var baseConstraints = {
         name: kwArgs.name || "",
         label: kwArgs.label || "",
         dataType: kwArgs.dataType,
         readOnly: kwArgs.readOnly,
         id: kwArgs.id,
         promptText: this.getPromptText(kwArgs),
         required: kwArgs.required
      };

      var editor = null;
      kwArgs.cardinality = kwArgs.cardinality.toUpperCase();
      if (kwArgs.cardinality && (kwArgs.cardinality == "LIST" ||
kwArgs.cardinality == "MULTI")) {
         editor = this._createMultiValueEditor(baseConstraints,
kwArgs);
```

```
      } else if (kwArgs.cardinality && kwArgs.cardinality == "SINGLE"
&& kwArgs.choiceList && kwArgs.choiceListNested) {
         editor = this._createSingleChoiceTreeEditor(baseConstraints,
kwArgs);
      } else if ((kwArgs.cardinality && kwArgs.cardinality ==
"SEARCHCHOICE" && kwArgs.choiceList)) {
         editor = this._createSearchChoiceListEditor(baseConstraints,
kwArgs);
      } else if ((kwArgs.cardinality && kwArgs.cardinality == "SINGLE"
&& kwArgs.choiceList) || (kwArgs.valueOptions &&
kwArgs.valueOptions.length > 0)) {
            editor =
this._createSingleChoiceListEditor(baseConstraints, kwArgs);
      } else if (kwArgs.dataType == "xs:date" || kwArgs.dataType ==
"xs:timestamp") {
         editor = this._createDateTimestampEditor(baseConstraints,
kwArgs);
      } else if (kwArgs.dataType == "xs:time") {
         editor = this._createTimeEditor(baseConstraints, kwArgs);
      } else if (kwArgs.dataType == "xs:guid") {
         editor = this._createGuidEditor(baseConstraints, kwArgs);
      } else if (kwArgs.dataType == "xs:reference") {
         editor = this._createReferenceEditor(baseConstraints, kwArgs);
      } else if (kwArgs.dataType == "xs:object") {
         editor = this._createObjectEditor(baseConstraints, kwArgs);
      } else if (kwArgs.dataType == "xs:integer" || kwArgs.dataType ==
"xs:short" || kwArgs.dataType == "xs:long" || kwArgs.dataType ==
"xs:decimal" || kwArgs.dataType == "xs:double") {
         editor = this._createNumberEditor(baseConstraints, kwArgs);
      } else if (kwArgs.dataType == "xs:group") { // user and group
         editor = this._createGroupEditor(baseConstraints, kwArgs);
      } else if (kwArgs.dataType == "xs:user") { // user
         editor = this._createUserEditor(baseConstraints, kwArgs);
      } else {
         editor = this._createTextEditor(baseConstraints, kwArgs);
      }

      if (baseConstraints.label && editor) {
         editor.set("title", baseConstraints.label);
         editor.set("alt", baseConstraints.label);
      }
      this.logExit(methodName);
      return editor;
   },
```

```
   /**
    * @private Creates the search choice list editor.
    */
   SinglePropertyEditorFactory.prototype._createSearchChoiceListEditor
= function(baseConstraints, kwArgs) {

      var availableData = kwArgs.choiceList;

      var SearchChoice = new SearchChoicePane({
         availableData: availableData,
         selectedValues: kwArgs.values,
         isTree: kwArgs.choiceListNested,
         editable: !kwArgs.readOnly,
         hasSorting: !kwArgs.uniqueValues,
         hideAvailableOnAdd: kwArgs.uniqueValues,
         allowDuplicateValues: !kwArgs.uniqueValues
      });
      lang.mixin(baseConstraints, {
         "label": SearchChoice.getLabel(),
         width: kwArgs.width || "",
         dropDown: SearchChoice
      });
      var editor = new ecm.widget.DropDownInput(baseConstraints);
      editor.set("value", SearchChoice.getValue());
      editor.startup();

      return editor;
   };
});
```

In this sample, there are two Dojo widgets, SearchChoicePane and ChoiceGrid for the filter choice list. The ChoiceGrid is included in SearchChoicePane. SearchChoicePane is called in `SampleSearchChoiceListPlugin.js` in step 4. Because the Dojo part is not the key point in this chapter and the space is limited, the detailed sample code for the Dojo widgets are not listed here.

## 7.10 Deployment and configuration

Before we can deploy the application to the application server, which is WebSphere Application Server for our sample, we must create the tables on the database. This database can be the same as the database that is used for the IBM Content Navigator to keep this scenario simple. If you want, you can also create a separate database or use another existing database. The required tables can be created and loaded with the attached DDL scripts, which are in the downloadable material for this book (see Appendix D, “Additional material” on page 535 for download information). Be sure to set the necessary permissions on the table if you want to reuse the configured database user from the IBM Content Navigator data source.

The next preparation step is to specify a JDBC data source that points to our database, if you created a separate database and you deploy on a different application server than the one IBM Content Navigator is deployed to. Otherwise we can reuse the data source definition we already created through the IBM Content Navigator configuration and deployment tool. If you need to create a new data source, follow the instructions in your application server documentation.

To deploy the custom EDS implementation on a WebSphere Application Server, complete the following steps:

1. Export the project as a WAR file to your desired location.
2. Open the WebSphere Administration Console and log in.
3. Select **Applications** → **New Application** → **New Enterprise Application.**
4. Point the installation path to the WAR file that you exported in step 1 and go through all the steps by clicking **Next**. Then, select the JDBC data source reference, for example `jdbc/CIWEBDS` and the deployment root `/customEDSService`.
5. Click **Finish** and **Save** to save the master configuration.
6. Select **Applications** → **Application Types** → **WebSphere enterprise applications**, and start the EDS application, if did not start automatically.
7. For validation test, enter the URL to the GetObjectTypesServlet:

   `http://<server_name>[:<port>]/customEDSService/types`

This deployed EDS must be linked to your IBM Content Navigator deployment where you register and configure the external data services support plug-in, which is named edsPlugin, to use the sample external data services.

Complete the following steps:

1. Load the `edsPlugin.jar` file. In the IBM Content Navigator administration tool, in the configuration options for the plug-in, define the URL or the file path to the `edsPlugin.jar` file:
   a. Enter the URL in the following sample format:

      `http://<server_name>:<port>/<ECMClient_installdir>/plugins/edsPlugin.jar`

   b. Enter the path to the file that is installed on the server, for example, use one of the following paths:
      - Microsoft Windows:

        `C:\Program Files\IBM\ECMClient\plugins\edsPlugin.jar`

      - Linux or IBM AIX® system:

        `/opt/IBM/ECMClient/plugins/edsPlugin.jar`

      The ECMClient folder is the IBM Content Navigator installation directory.
2. After the `edsPlugin.jar` file is loaded, specify the URL to where the EDS is deployed. For example, you can specify the following URL:

   `http://<server_name>[:<port>]/customEDSService`
3. Click **Save** and **Close**.
4. Refresh your web browser.

Now your IBM Content Navigator has loaded the EDS plug-in. The plug-in is configured to call your deployed sample EDS service, and use the data that is returned to manipulate the data and properties that are displayed.

To register the request and response filter plug-in with IBM Content Navigator, complete the following steps:

1. Build the plug-in projects and copy the JAR file to the location you want.
2. Load the `SampleFilterPlugin.jar and SampleSearchChoiceListPlugin.jar` files. In the IBM Content Navigator administration tool, in the configuration options for the plug-in, input the file path to the plug-in JAR files and click **Load**.
3. Click **Save** and **Close**.
4. Refresh your web browser.

Now the request and response filter plug-in is loaded. The property order should be in the plug-in setting. The filter dialog should also be opened with the choice list property that is set in the plug-in.

### Performance considerations

When you want a custom EDS implementation and deployment, consider the following aspects:

- Put the EDS service physically as close as possible to the data that EDS is supposed to access; having it on the same machine, with no network between is the best approach to avoid any latency issues.
- Think about implementing a caching technique so that the response time for frequently requested object types is as low as possible.
- Scale the WebContainer settings of the application server thread, appropriate to the expected number of users and actions that call EDS.

## 7.11 Conclusion

This chapter describes request and response filters and one of the special implementations, EDS. EDS is a powerful extension that can be used to implement your custom requirements independently, based on a REST service syntax. EDS can control the property data and behavior. This chapter shows how to do an EDS implementation and several main features to manipulate the IBM Content Navigator user interface. It also shows how to use request and response filters to implement other features.

8

# Creating a custom step processor

This chapter describes how to create custom workflow step processors using the IBM Content Navigator toolkit. It provides background information about step processors and the step processors delivered with IBM Content Navigator. It then steps through the process of creating custom step processors.

This chapter covers the following topics:

- Workflow step processors overview
- IBM Content Navigator step processors
- Custom step processors
- Example overview
- Creating a custom step processor
- Deploying the custom step processor
- Registering the custom step processor
- Configuring Application Space and in-basket
- Adding an action to use an embedded viewer
- Using and testing the custom step processor
- Adding external data services

© Copyright IBM Corp. 2012, 2014. All rights reserved.

## 8.1 Workflow step processors overview

A *workflow* is a series of work steps connected by routes. The routes determine which step is processed next based on the output from the previous step. The steps perform the actual tasks of the workflow.

Workflow steps can be categorized into work performers and step processors. A *work performer* is an automated task that contains no user interface. Work performers perform some action based on the information in the workflow. This action can include updating the attached documents, calling external systems, performing IBM FileNet P8 operations or other related activities. A *step processor* is used to display a user interface and is added to a workflow queue for processing. When the step is executed by a user, the user interface is displayed and the user will perform the step. After the user has completed the appropriate actions, the user completes the step and the workflow uses the routes from that step to determine the next step to execute.

When designing a workflow, the designer adds a step processor to the workflows by dragging the Activity icon onto the design window. They then configure the step processor through the properties user interface. The first step is to specify which step processor to use. If IBM Content Navigator is installed and configured, then the IBM Content Navigator step and launch processors will appear in the list of available step processors. Additionally, any custom step processors will appear if they have been registered.

## 8.2 IBM Content Navigator step processors

IBM Content Navigator provides two configurable step processors for use in workflows. They are a launch processor and a step processor. The *launch processor* is a special type of step processor that is used as the first step in a workflow and is used to start the workflow process. A launch processor can only be used as the first step in the workflow. The *step processor* can be used anywhere in a workflow except as the first step. The step processor is used to provide the user interface for workflow steps that require user interaction.

The step processors delivered with Content Navigator provide functionality that will work for many uses cases. They provide a multi-tab user interface that displays the information needed for the user to complete the step. On the first tab, all of the properties configured for the step are displayed. The user can examine existing values and input values required to complete the step. Figure 8-1 on page 279 shows a sample of the Properties tab.

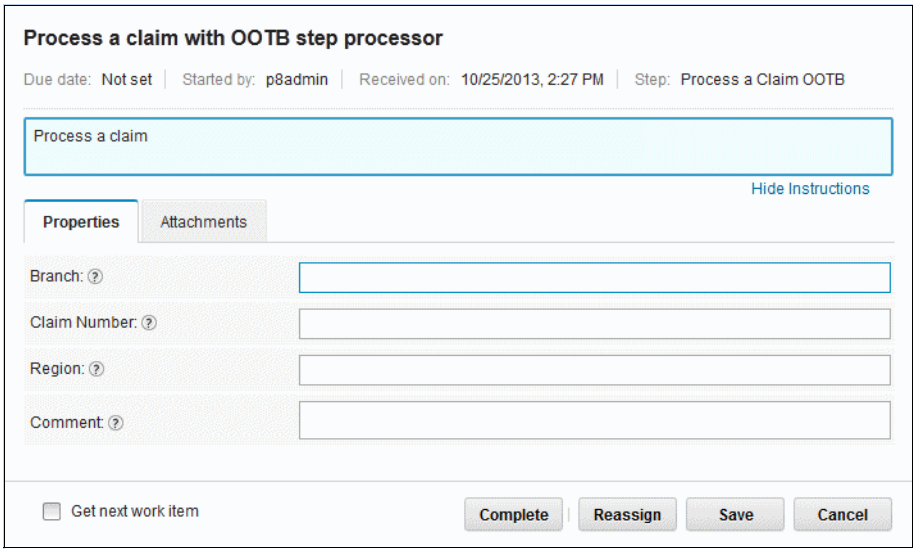



*Figure 8-1 Content Navigator step processor properties page*

The second tab displays any documents that have been attached to the workflow and that are configured to display. The user can add additional documents and view the existing documents. Figure 8-2 shows the Attachments tab of the Content Navigator step processor.

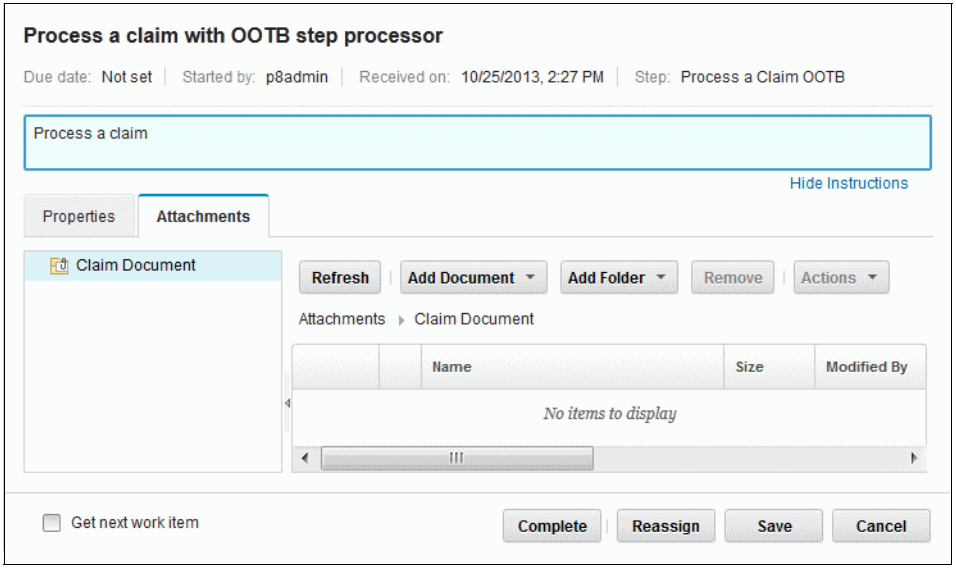



*Figure 8-2 Content Navigator step processor attachments page*

There is also an optional third tab that will show the history of the workflow. In addition, these step processors provide buttons to perform the various workflow step actions such as complete the step, save the work in progress, cancel, or reassign the work to someone else.

The Process Designer is the tool that is used to create and modify workflows. It allows the users to drag workflow steps onto the design space, configure the steps, and link them to other steps. When you add the Content Navigator step processor to a workflow in the Process Designer, several basic customizations are available that can be done through simple configuration. To perform these customizations, add an activity to the workflow in the Process Designer and then select this activity. Then select **Navigator Step Processor** from the Step Processor dropdown list. This specifies that the Content Navigator step processor is used for this step.

The first customization is the Instructions that are presented to the user. The Instructions inform the user of the details of the task to be performed. It provides the information they need to complete the task.

When configuring the step processor, you can also decide which properties are displayed to the user. On the Parameters tab in Process Designer, you can select which of the workflow properties are displayed to the user. In the step processor itself, these will show up on the Properties tab and will show any values that have already been entered. You can enter a prompt that is displayed for each property and you can also specify whether the properties are read-only or read-write. The step processor will sort the properties displayed based on the prompt specified in Process Designer. It will then add any workflow group properties at the bottom of the standard fields. Finally, the Comments field will be displayed last.

You can select which workflow attachments will be included in the step processor. If there is a specific attachment that is needed, you add it to the list of parameters. You can include all of the attachments, some of the attachments or none of the attachments in the step processor. When defining the workflow attachments in the Process Designer, you can designate one of the attachment parameters as the Initiating Attachment by select the attachment in the Attachments tab and clicking the Initiating Attachment button at the top.

You can also specify whether the user processing the step can reassign the work to another user, whether the History tab is displayed and whether the workflow status is displayed. These options are configured on the General tab of the step processor configuration page.

## 8.3 Custom step processors

The step processor delivered by Content Navigator will cover many of the use cases for your work steps. If you need additional control of the step processor then you can create a custom step processor. With a custom step processor, you modify the Content Navigator step processor user interface by adding, or modifying the widgets that are shown in the user interface.

The Content Navigator step processors are created using the Content Navigator toolkit. When you create a custom step processor you create a new step processor that extends the delivered step processor. When you extend the delivered step processor, you start with the functionality that it provides, but you can then extend it to include additional user interface components or to provide additional functionality. Additionally, you can integrate Content Navigator external data services for the properties that are displayed in the custom step processor.

## 8.4 Example overview

This example will create a custom step processor to process claims introduced in earlier chapters. When a document is added to the system and assigned to the Claims document class, an event action will be triggered that will start a claims processing workflow. This workflow will utilize a custom step processor to review and approve the claim.

In the Content Navigator step processor, you can view an attached document by opening the document in the attachments pane. When you open the document, the Content Navigator viewer opens in a separate browser window to display the selected document. The custom step processor will embed the Content Navigator viewer into the step processor user interface. This viewer will be in the same browser window as the step processor instead of in a separate browser window like the Content Navigator step processor.

Additionally, the external data services developed in Chapter 7, “Implementing request and response filters and external data services” on page 233 will be used to control the properties that are displayed in the step processor user interface. The claim number validation will be used and the dependent property lists for the Region and Branch fields.

## 8.5 Creating a custom step processor

The sample custom step processor will extend the delivered Content Navigator step processor. This will allow the custom step processor to retain the functionality provided by the Content Navigator step processor while adding the new functionality for our sample. The existing step processor widget is being extended, so only the extensions we are providing need to be coded. The other functionality will be deferred to the existing step processor.

Three files implement the Content Navigator step processor:

`StepProcessor.jsp located in the WebContent directory`
`StepProcessorLayout.html located in ecm\widget\process\templates`
`StepProcessorLayout.js located in ecm\widget\process`

For this example, three new files will be created that will be deployed to Content Navigator. The following sections describe these files and the code that will be implemented in each.

### 8.5.1 StepProcessorRedbkLayout.html

This file provides the HTML template for the step processor widget. To create this file, make a copy of the `StepProcessorLayout.html` delivered by Content Navigator and name it `StepProcessorRedbkLayout.html`. This template will be modified to add a new container for the viewer that will be included in the custom step processor. The layout will include the existing tab container for the step processor on the left and will add the viewer to the right side of the tab container.

The first step is to modify the existing stepContentPane. In Content Navigator processor, this pane uses the full width of the step processor widget. This must be modified so that the dialog will be split between this pane and the viewer pane that is being added. To do this, you first change the region from `center` to `leading`. This will place this widget on the left. Next, you add the following attributes to this pane:

```
splitter="true"
style="width: 50%"
```

This adds a splitter between the stepContentPane and the viewer pane and it specifies that the stepContentPane will be allocated half of the width of the widget.

Example 8-1 shows the updates for the content pane in `StepProcessorRedbkLayout.html`.

*Example 8-1 Update content pane in StepProcessorRedbkLayout.html*

```
<div data-dojo-type="dijit.layout.ContentPane"
   data-dojo-attach-point="stepContentPane"
   region="leading" splitter="true" style="width: 50%" >
```

Next, the viewer container is added to the template after the stepContentPane and before the ActionBar pane, which is at the bottom of the widget. This container will hold the viewer widget that is added. Example 8-2 shows the code to add the content pane.

*Example 8-2 Add ContentPane to StepProcessorRedbkLayout.html*

```
<div data-dojo-attach-point="customViewerContainer"
      data-dojo-type="dijit.layout.ContentPane"
      region="center" style="width: 50%" >
   <div id="contentViewer"style="width: 100%; height: 100%">
   </div>
</div>
```

## 8.5.2 StepProcessorRedbkLayout.js

StepProcessorRedbkLayout will create the step processor widget. This widget extends the Content Navigator step processor so it includes only the additional code needed by the widget to include the viewer. The base functionality of the widget will be provided by the widget that is being extended, which is defined in the `StepProcessorLayout.js` file.

This file starts by defining the files needed by the widget and then declares the widget that is being creating. The code of most interest in this file is the startup function, which will initialize the viewer. This procedure checks if the viewer has already been created by checking contentViewer. If the viewer has not been created, then this procedure creates an instance of the Content Navigator viewer and assigns it to contentViewer. Example 8-3 shows the code for `StepprocessorRedbkLayout.js`.

*Example 8-3 Listing for StepProcessorRedbklLayout.js*

```
define([
   "dojo/_base/declare",
   "dojo/_base/lang",
   "dojo/_base/connect",
   "ecm/LoggerMixin",
```

```
   "ecm/widget/process/_ProcessorMixin",
   "ecm/widget/process/StepProcessorLayout",
   "dojo/text!./templates/StepProcessorRedbkLayout.html"
], function(declare, lang, connect, LoggerMixin, _ProcessorMixin,
StepProcessorLayout, template) {
   /**
    * @name custom.widget.process.StepProcessorRedbkLayout
    * @class Provides a custom layout for step processors.
    * @augments custom.widget.process.StepProcessorLayout
    */
   return declare("custom.widget.process.StepProcessorRedbkLayout", [
      StepProcessorLayout
   ], {
      /** @lends custom.widget.process.StepProcessorRedbkLayout.prototype */

      // widgetsInTemplate: Boolean
      //    Set this to true if your widget contains other widgets
      widgetsInTemplate: true,
      contentString: template,

      contentContainer: null,
      contentViewer: null,

      // postCreate() is called to override the superclass.
      postCreate: function() {
         this.logEntry("postCreate-custom");
         this.inherited(arguments);
         this.logExit("postCreate-custom");
      },

      // startup() is called to create the create and place the
      // Content Viewer instance, resize the Content Viewer layout
      // and override the default Viewer toolbar text.
      startup: function() {
         this.logEntry("startup-custom");
         this.inherited(arguments);

         // if an instance of the Content Viewer doesn't exist then create one
         if (this.contentViewer == null) {
            var bidiDir = "ltr"; // left-to-right for English locale as
default.
            var language = dojo.locale;
            if ((language.indexOf("ar") === 0) || (language.indexOf("he") ===
0) || (language.indexOf("iw") === 0)) {
               bidiDir = "rtl"; // Use right-to-left for Arabic locale.
            }

            dojo['require']("ecm.widget.viewer.ContentViewer");
```

```
            var tabStyle = "margin: 0px; padding: 0px; width: 100%; height:
100%;";

            // create an instance of the Content Viewer
            this.logDebug("startup-custom", "instantiate CV");
            this.contentViewer = new ecm.widget.viewer.ContentViewer({
               style: tabStyle,
               isEntireWindow: false,
               dir: bidiDir,
               lang: language
            });

            // place the Content Viewer on the step processor page
            this.logDebug("startup-custom", "place CV in the step processor
page");
            this.contentContainer = dojo.byId("contentViewer");
            this.contentContainer.appendChild(this.contentViewer.domNode);

            // resize the Content Viewer layout
            this.logDebug("startup-custom", "resizing the CV layout");
            this.contentViewer.startup();

            // set the text on the Content Viewer toolbar
            this.logDebug("startup-custom", "set the text on the CV toolbar");
            this.contentViewer.viewerToolbarText.innerHTML = "Viewer";

            // detect splitter movement and resize container
            this.connect(this.stepContentPane._splitterWidget.domNode,
"onmouseup", function() {
               if (this.contentViewer != null) {
                  this.contentViewer._winResizeEnd();
               }
            });
         }

         this.logExit("startup-custom");
      },

   });
});
```

## 8.5.3 StepProcessorRedbkLayout.jsp

The final file is StepProcessorRedbk.jsp. This is the step processor controller and will be registered with the Process Configuration Console to identify it as a step processor that can be used in workflows.

This file will be based on the StepProcessor.jsp file with only minor changes so that it uses the custom step processor widget instead of the step processor widget delivered with Content Navigator.

First, copy StepProcessor.jsp and name the copy StepProcessorRedbk.jsp. Next, add dojo.require for the custom widget so that it can be used. Then, update the controller to use the custom step processor. Example 8-4 shows the code for StepProcessorRedbk.jsp.

*Example 8-4 Listing for StepProcessorRedbk.jsp*

```
<!DOCTYPE html>
<%
    String htmlLocale =
request.getLocale().toString().toLowerCase().replace('_','-');
    htmlLocale = htmlLocale.replace("iw", "he");  // workaround for
Java getLocale() spec.
%>
<html lang="<%=htmlLocale%>">
<%
    response.setHeader("Cache-Control","no-cache");
    response.setHeader("Pragma","no-cache"); //HTTP 1.0
    response.setDateHeader ("Expires", 0); //prevents caching at the
proxy server
%>
<%@ include file="header.jsp" %>

<body class="<%=bodyClasses%>" dir="<%=direction%>" style="width: 100%;
height: 100%; position: absolute;">
    <div id="ECMWebUIloadingAnimation"
style="display:inline-block;position:absolute;top: 10px; left: 10px">
        <div class="ecmLoading"></div>
        <div id="ECMWebUIloadingText" class="contentNode
loadingText"></div>
    </div>
    <script>
   dojo.require("custom.widget.process.StepProcessorRedbkLayout");
    </script>
    <div dojoType="custom.widget.process.StepProcessorRedbkLayout"
id="ECMStepProcUI" style="width: 100%; height: 100%"
lang="<%=htmlLocale%>"></div>
</body>

</html>
```

## 8.6 Deploying the custom step processor

The implementation files must be installed directly into the Content Navigator deployed directories. To deploy the custom step process, complete these steps:

1. Stop the Content Navigator application server.
2. Copy the custom step processor into the Content Navigator deployed location on the application server.
3. Start the Content Navigator application server.

Copy the files into the following locations:

- Copy `StepProcessorRedbk.jsp` into the Content Navigator `WebContent` directory.
- Under `WebContent`, create the directory `custom\widget\process` and copy `StepProcessorRedbkLayout.js` in the directory you created.
- Create a templates directory under the custom directory you created and copy `StepProcessRedbkLayout.html` into the templates directory.

## 8.7 Registering the custom step processor

Before the custom step processor can be used in a workflow, it must be registered through the Process Configuration Console. Registering the step processor makes it available in the Process Designer as one of the supported step processor. Additionally, registering the step processor tells the Content Platform Engine where to find the custom step processor when a step is executed.

To register the custom step processor in the Process Configuration Console, complete the following steps:

1. In the tree view on the left, select the context menu for the appropriate connection point and select **Connect**.
2. After you are connected, select the context menu for the connection point again and select **Properties**.
3. Click the **Web Applications** tab and ensure that the appropriate URL has already been specified for Content Navigator.
4. Click the **Step Processor Info** tab.

5. Click the **Add** icon in the upper right side and enter the following information and configuration:
   - Enter `Step Processor Redbk` as the name for the custom step processor.
   - For the language, select **HTML**.
   - Update the width to `1200` and the height to `800`. With the embedded viewer, a larger dialog is needed.
   - Enter location information:
     i. Double click in the **Location** field which will display the Step Processor Locations dialog.
     ii. Select the **IBM Content Navigator** entry.
     iii. Enter `StepProcessorRedbk.jsp` into the Location field.
     iv. Click **OK** to close the Location dialog box.
6. Click **OK** again to close the Add dialog box and commit the changes.

## 8.8 Configuring Application Space and in-basket

Content Navigator uses Application Spaces and in-baskets for displaying the list of work items to be processed. If you do not already have one, then you need to create a new Application Space in the Process Configuration Console.

To create the new Application Space, complete these steps:

1. In Process Configuration Console. Select the context menu for **Application Spaces** and select **New**.
2. In the dialog, enter a name and description and then press **OK**.

To configure the Application Space, complete these steps:

1. Open the created Application Space.
2. Click the **Roles** tab, add a new role and name it `ProcessClaims`.
3. Under Select in-baskets, select the **inbox** from the list of in-baskets.
4. Select the users that will be processing the claims.
5. Save the changes by pressing **OK**.

The Application Space is then displayed in the Content Navigator Work View. When the Application Space is expanded, you can select an in-basket, and then the work items that are assigned to it will be displayed. The work item can be opened by double-clicking it or by selecting it and clicking **Open**.

## 8.9 Adding an action to use an embedded viewer

The next step is to provide a method for users to display documents in the embedded viewer. The Content Navigator step processor will automatically display the document in the external viewer, so an action is added that allows the document to be displayed in the embedded viewer. The default Content Navigator behavior will not be overridden so the user will have the option of using the embedded or non-embedded viewer.

To add the action, a Content Navigator plug-in is created by using the Content Navigator Eclipse extensions introduced in Chapter 3, “Setting up the development environment” on page 73. Follow the instructions in that chapter to create a new Content Navigator plug-in project. Table 8-1 shows the values to enter when creating the project.

*Table 8-1 Values for creating a new plug-in*

| Field | Value |
|---|---|
| Project name | StepProcessorActionPlugin |
| Descriptive name | Step Processor Action Plugin |
| Java package | com.ibm.ecm.icn.plugin |
| Class name | StepProcessorAction |
| Version | 1.0 |

This plug-in is needed to deliver the action. Follow the instructions in Chapter 5, “Building a custom repository search service” on page 173 to create a plug-in extension. In this case, an action is added, so select **Action** from the list of extensions. Enter `com.ibm.ecm.icn.plugin` as the Java Package and `StepProcessorAction` as the Class Name.

To implement the action, several files that are generated must be modified. The following sections outline the necessary changes.

### 8.9.1 Updating StepProcessorAction.java

The `StepProcessorAction.java` file provides some general information about the action that is being implemented. Information specific to the action will need to be added to several of the procedures.

The getName() method specifies the string that will be displayed on menus for the new action. Update this field to a “user-friendly” string that describes the action. Example 8-5 shows the code for updating this procedure.

*Example 8-5 Listing for getName method*

```
public String getName(Locale locale) {
   return "Open in Embedded Viewer";
}
```

The getPrivilege() method must be updated to specify which privilege is needed to perform this action. Anyone who can view the document will be able to perform this action, so this procedure will return the value privViewDoc. Example 8-6 shows the code for the getPrivilege() method.

*Example 8-6 Listing for getPrivilege method*

```
public String getPrivilege() {
   return "privViewDoc";
}
```

The getServerTypes() method must return the types of servers that are supported. This example will be working with IBM FileNet P8, so return `"p8"` as the value. Example 8-7 shows the code for the getServerTypes() method.

*Example 8-7 Listing for getServerTypes method*

```
public String getServerTypes() {
   return "p8";
}
```

### 8.9.2 Updating StepProcessorActionPlugin.js

The `StepProcessorActionPlugin.js` file performs the action that is being added. It verifies that a document has been selected, and then verifies that the embedded viewer has been created. If both of these conditions are met, then it sends the document to the embedded viewer. Example 8-8 on page 291 shows the code for the `StepProcessorActionPlugin.js` file.

*Example 8-8 Listing for StepProcessorActionPlugin.js*

```
require(["dojo/_base/declare",
         "dojo/_base/lang",
         "ecm/widget/viewer/ContentViewer" ],
function(declare, lang, ContentViewer ) {
    lang.setObject("stepProcessorAction", function (repository, items,
callback, teamspace, resultSet, parameterMap) {
    // action definition only permits one object in the array
    var item = items[0];
    // check if the item is a document
    if (!item.isFolder()) {
        // check if viewer instance exists
        if (window.contentViewer != null) {
            // open the document in the viewer
            window.contentViewer.open(item);
        }
    }
});
});
```

## 8.9.3 Building and deploying the action plug-in

The project generated by the Eclipse tooling automatically creates a `build.xml` file to build the plug-in JAR file. Run `build.xml` to create the plug-in JAR file. After the JAR file is created, register and test it by following the instructions in 3.3.4, “Registering and testing a plug-in” on page 94.

After the plug-in is registered, you can add the action provided by the plug-in to the appropriate menu:

1. Open Content Navigator with administrator privileges and open the Administration View feature.
2. Expand **Menus** in the tree and find the **Default attachment item context menu**. The default menu cannot be modified, so a copy of the menu must be created.
3. Select the context menu for the menu in the list and select **Copy**. This will create a new menu of the same type as the menu selected.
4. Enter a name for the custom menu in the dialog. The ID will automatically be populated based on the name you specify.
5. From the Available Actions on the left, select the **Open in embedded viewer** action from the plug-in, and move it to the Selected Actions area.
6. Save the new menu that was created.

Next, assign the new menu to the appropriate toolbar:

1. Open the default desktop (or whichever desktop will be used), and select the **Menu** tab.
2. Under the FileNet P8 Workflow Context Menus section, select the drop-down next to the **Document Attachment Context Menu** and select the custom menu that was created previously.
3. Save the changes to the desktop.

## 8.10 Using and testing the custom step processor

To use and test the custom step processor, create a workflow that incorporates it. The workflow is created in the Process Designer. A subscription is used to initiate the workflow whenever a document is added for the Claim document class. The following sections provide an overview of creating and testing the workflow.

### 8.10.1 Creating the workflow with custom step processor

This example creates a simple workflow that consists of a launch step and then an activity step that uses the custom step processor. It will include a set of properties to display in addition to a single document attachment. The following steps describe how to create the workflow:

1. Open Process Designer. It opens with a new workflow that contains a launch step.
2. In the Workflow tab at the bottom, enter `Claim Processing` as the Workflow Name. Enter `Process a claim` as the subject. Optionally, enter a description.
3. On the Data Fields tab, add three string data fields, using these names for them: `Branch`, `ClaimNumber`, and `Region`. Enter a description for each data field.
4. On the Attachments tab, add an attachment named `ClaimDocument`. Leave the Array and Value fields blank, but enter a description.
5. Select the **LaunchStep** in the graphic area. On the General tab, enter `LaunchClaim` as the Step Name.
6. Add an activity step by dragging the **Activity** icon to the graphic area.
7. On the General tab for the activity, do these tasks:
   - Enter `ProcessClaim` as the Step Name.
   - In the participants field, add **F_Originator**.

- In the Step Processor drop-down list, select **Step Processor Redbk step processor**. If this step processor is not listed, recheck the configuration of the step processor in the Process Configuration Console.
- Enter instructions explaining to the user how to process the claim in this custom step.
- On the Parameters tab, add all four parameters to the list of selected parameters. Enter a prompt for each parameter.

8. Add a route from the LaunchStep to the custom step.

### 8.10.2 Adding subscription to initiate workflow

The sample workflow can be launched manually, or to automate the process you can create a workflow subscription. For this example, create a subscription that launches the workflow when a document in the Claim document class is created. This is done by subscribing to the check-in event for Claim and launching the workflow. This way, whenever a document in this class is created in the object store, the subscription is automatically launched with the new document as an attachment. The workflow will then move to the Process Claim step and a work item will arrive in the appropriate in-basket to process the claim.

## 8.11 Adding external data services

External data services can be configured to work with workflow step processors including custom step processors. For this example, the external data services are created in Chapter 7, “Implementing request and response filters and external data services” on page 233. Through a simple modification, the external data services that were created can be added to the custom step processor.

The external data services were configured to work on the Claim document class and provided several customizations. EDS is used for the following fields:

- ClaimNumber: Validate the format of this field.
- Region: Add a choice list.
- BranchOffice: Makes this choice list a dependent property of Region.

You will notice that the field names used in the custom step processor match the field names from the Claims document class. This allows you to reuse the EDS that was created for the document class fields.

To use these items, the external data services must be updated to include the workflow step processor as a target. This is done by updating the `ObjectTypes.json` file in the external data services project to include the

workflow in the list of supported classes. Example 8-9 shows the updated code. In the example, Claim_Processing is the workflow name, spaces are replaced by underscores, and ProcessClaim is the step name in the workflow.

*Example 8-9 ObjectTypes.json updated to add the custom step processor*

```
[
{"symbolicName": "Claim"},
{"symbolicName": "Claim_Processing.Workflow.ProcessClaim"},
]
```

After this change is made, regenerate the WAR file for the EDS service and redeploy it to the application server. This simple change enables EDS for the step processor.

External data services can provide a variety of functionality to validate and simplify the entry of data by the user. As was shown previously, you can create choice lists, designate dependent choice lists, and validate the data entered by the user. Other key features that can be provided by external data services include providing default values, defining minimum and maximum values, and specifying the length of a string field. If a string field is specified as 256 characters or more, then the entry area for that field will automatically become a multi-line input box.

## 8.11.1 Testing the workflow and custom step processor

The Process Designer tool can save and launch your workflow, to help you more easily perform tests.

To test the workflow and custom process step processor, use these steps:

1. Select **File** → **Launch Main Workflow** from the Process Designer menu.
2. In the first window, select the object store and folder where the workflow will be saved and then click **Next**.
3. On the Save the Workflow dialog, enter `ICN Claim Processing` as the Document Title and click **Finish**. This saves and then launches the workflow.
4. On the Launch dialog, click **Launch**.
5. Start Content Navigator, select the Process feature and then Navigate to the in-basket you created from the Process Configuration Console. The listview contains a work item for the workflow that you launched.
6. Open the work item to view the custom step processor.

Figure 8-3 shows the custom step processor property pages. Notice that the validation of the claim number is being performed and that the region and branch are now associated with choice lists.

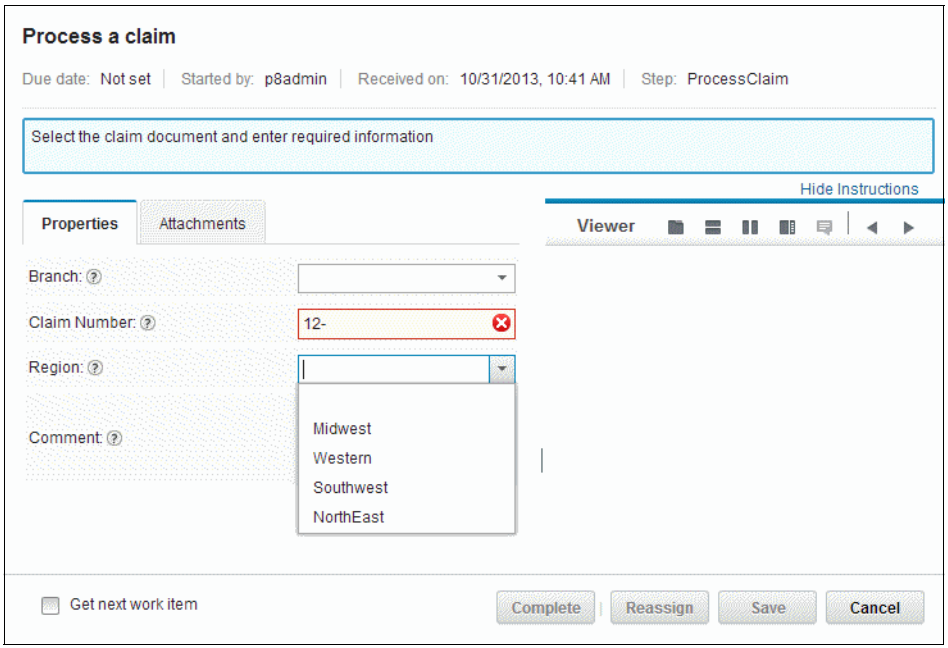



*Figure 8-3 Custom step processor properties tab*

Figure 8-4 shows the custom step processor attachments page with a document selected and displayed in the embedded viewer.

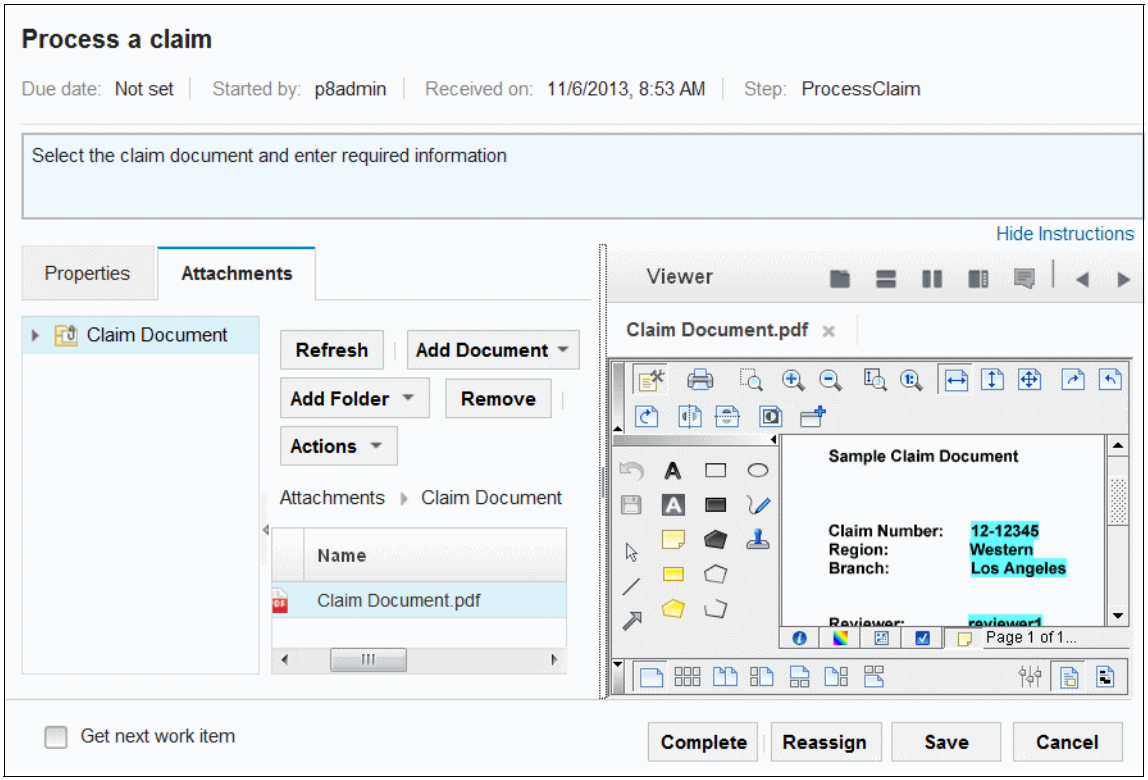



*Figure 8-4 Custom step processor attachments tab*

## 8.12 Conclusion

This chapter shows how to create a custom step processor based on the Content Navigator step processor. The custom step processor modifies the original step processor to provide an embedded viewer for displaying documents associated with the workflow. This example shows how you can create custom step processors to meet the functional needs of your application.

9

# Using Content Navigator widgets in other applications

The chapter describes how IBM Content Navigator components can be integrated into external applications. An external application can be a stand-alone application that invokes components of IBM Content Navigator or it can be a container like a portal server that provides the runtime environment in which IBM Content Navigator components must be executed.

This chapter covers the following topics:

- Integration into other applications overview
- Example overview
- Externalizing IBM Content Navigator widgets
- Integrating Content Navigator with URL API
- Integrating Content Navigator with a specific feature
- Integrating Content Navigator with a specific layout
- Integrating specific widgets of Content Navigator
- Integrating the externalized widgets (Step 5)
- Integrating stand-alone widgets in a Microsoft SharePoint page
- Integrating stand-alone widgets as a portlet in WebSphere Portal

© Copyright IBM Corp. 2012, 2014. All rights reserved.

## 9.1 Integration into other applications overview

When integrating IBM Content Navigator or only some of its components into an external application, you typically must complete two steps:

1. Externalize IBM Content Navigator widgets so that only the part that needs to be integrated is displayed.

   This can be either the entire IBM Content Navigator application or only one or some of its widgets. It can be the standard layout with the “look and feel” of IBM Content Navigator or a custom layout with a different style.

   This step can be accomplished independently of the final target system in which the IBM Content Navigator should be integrated and is covered in the first part of this chapter.

2. Integrate the externalized IBM Content Navigator parts from step 1 into the external application.

   This can be a simple URL invocation of IBM Content Navigator, for example in an iFrame of the external application, or it can be a tighter integration where the JavaScript code is running in the external application. This, of course, depends on the target system.

## 9.2 Example overview

This chapter shows examples of externalizing IBM Content Navigator widgets and the following integrations:

- Integrating Content Navigator with URL API
- Integrating Content Navigator with a specific feature
- Integrating Content Navigator with a specific layout
- Integrating stand-alone widgets in a Microsoft SharePoint page
- Integrating stand-alone widgets as a portlet in WebSphere Portal

In principle, the steps for integrating into other systems are similar.

**Note:** The first part of the chapter provides several approaches for how to externalize IBM Content Navigator or parts of it. In a real-world scenario, you do not exercise all of the approaches; rather you choose the most appropriate approach, which is probably the simplest one that complies with all the requirements for your scenario.

The order of the approaches starts with the simplest and least flexible approach and progresses to the most complex but most flexible approach.

# 9.3 Externalizing IBM Content Navigator widgets

The first part of integration is to prepare IBM Content Navigator to show exactly the part that you want to integrate into the external application.

We introduce various approaches of externalizing IBM Content Navigator and provide a web application that is used to simulate the integration to the target system.

## 9.3.1 Approaches of IBM Content Navigator externalization

Depending on the requirements, there are several approaches with a different level of integration. This section provides an overview.

The approaches can be categorized into *unbound* and *bound* integration as introduced in 1.2.4, “Developing or integrating a custom application” on page 11.

- *Unbound integration* means the external application is, in some way, invoking a URL of IBM Content Navigator application and the IBM Content Navigator widgets are running outside the external application.
- *Bound integration* means initializing, invoking, and interacting with the IBM Content Navigator widgets directly and not through URL invocation. IBM Content Navigator widgets are run inside the external application or container.

### Unbound integration

Unbound integration includes these features:

- Externalize with URL API and deep linking
- Externalize with your own custom feature
- Externalize with your own layout

#### *Externalize with URL API and deep linking*

Full IBM Content Navigator is integrated as a whole application. From the external application, you basically invoke the URL of IBM Content Navigator. With additional URL parameters provided by the URL API, you gain certain control regarding what parts of IBM Content Navigator should be displayed. This approach is described in 9.4, “Integrating Content Navigator with URL API” on page 308. If you need more flexibility and control, you must choose one of the other approaches.

Advantages and disadvantages of this approach are as follows:

- Advantages:
  - No coding, only configuration.
  - iFrame integration: no cross-site scripting problem.
  - No Dojo version conflicts between external application and IBM Content Navigator.
- Disadvantages:
  - iFrame integration:
    - Limited interaction with the master application (by changing the URL).
    - Each invocation will always reload the whole desktop.
  - Integrates whole IBM Content Navigator (not just a specific part or several widgets).

### *Externalize with your own custom feature*

If the first approach is not enough and you want to externalize something that is not covered by the default features of IBM Content Navigator, consider providing your own feature through a plug-in and invoking this feature directly with the URL API. For example, if you want a specific feature but without the bars at the top and left sides, you can additionally specify the `sideChrome` parameter. This approach is described in 9.5, “Integrating Content Navigator with a specific feature” on page 310.

This approach is already powerful. Developing your own feature gives you total freedom regarding which widgets to display and how to arrange them on the page. All IBM Content Navigator widgets and models can be used or you can develop your own widgets. There is practically no limitation if you want to show your custom feature. But to change the default layout of the IBM Content Navigator or to externalize only a specific widget and not launch the whole IBM Content Navigator with the default launch page, consider other approaches.

Advantages and disadvantages of this approach are as follows:

- Advantages:
  - You can specify which widgets to render in a custom feature.
  - Seamless integration is possible with the `sideChrome` parameter.
  - iFrame integration: no cross-site scripting problem.
- Disadvantages:
  - iFrame integration:
    - Limited interaction with the master application (by changing the URL).
    - Each invocation will always reload the whole desktop.
  - You cannot change the top banner or left feature bar (only show or hide).

### *Externalize with your own layout*

Like the first two approaches, you start the whole IBM Content Navigator through URL invocation of the standard launch page. But this time you change the standard layout of IBM Content Navigator (which is NavigatorMainLayout widget) with a custom layout that is provided with a plug-in. In this way, you can arrange features as you need or even abandon features. This approach is described in 9.6, “Integrating Content Navigator with a specific layout” on page 313.

If you want to provide a custom page with widgets of your choice and arrange them to your needs, this approach is as powerful as the previous one that uses a custom feature to achieve the same goal. To eliminate disturbing banners or bars, you basically skip them in your custom layout; however, in the custom feature approach, you can specify the sideChrome parameter and have the same result.

You have more flexibility than with the previous approach if you need to change aspects of the default layout. For example, perhaps the external application displays other features in a tab container than the button bar on the left side, like the IBM Content Navigator is doing in the standard layout. Then, consider developing a custom layout where the user can switch the features of IBM Content Navigator in a tab container to provide the same user experience with the legacy application that the user is familiar with.

Advantages and disadvantages of this approach are as follows:

- Advantages:
  - You can specify which widgets to render.
  - You can customize the top banner or the left feature bar.
  - Layout can contain several features and act as a container for all your “integration” features.
  - iFrame integration: no cross-site scripting problem.
- Disadvantages:
  - iFrame integration:
    - Limited interaction with the master application (by changing the URL).
    - Each invocation will always reload the whole desktop.

## Bound integration

This integration differs from the previous approach in that it uses widgets outside the standard IBM Content Navigator application and the code runs in the external application. You prepare the widgets you want to externalize and you do not launch the entire IBM Content Navigator with its standard initialization process.

This approach is described in 9.7, “Integrating specific widgets of Content Navigator” on page 324.

This approach provides the most flexibility. Your are not restricted to the IBM Content Navigator application. You have total freedom about which widgets to display, how to initialize them, and how to arrange them. You also have a deeper level of interaction with the external application because it’s not just invoking one single URL but rather wiring arbitrary events and provide interaction between components of the application and the externalized widgets of IBM Content Navigator.

But you must have a good understanding on how IBM Content Navigator is working internally, for example how the models are loaded and initialized and how to wire the widgets you want to externalize.

One of the key preconditions for this integration type is that Dojo framework is supported on the target system and that the exact Dojo version, which is used by IBM Content Navigator, can be executed inside the external application.

Advantages and disadvantages of this approach are as follows:

- Advantages:
  - It allows interaction with the master application and wiring with its components.
  - You can load the desktop model of IBM Content Navigator once.
  - You can control of the initialization process of the widgets to suit your needs.
- Disadvantages:
  - Requires a deep understanding of the IBM Content Navigator and its wiring.
  - Requires more coding and initialization of the javascript Model.
  - Confronts you with cross-site scripting.
  - Might lead to Dojo version conflicts if the external application also uses Dojo framework.

## 9.3.2 Simulation of IBM Content Navigator integration part 1

To test the first part of the integration, we simulate the second part, which is the invocation of the IBM Content Navigator from an external application: We provide a simple web application with a simple welcome page. This page is where we integrate IBM Content Navigator. We can emulate unbound and bound integration.

Figure 9-1 illustrates an unbound integration. The Container Simulation is a small web application that can be deployed to the same web server where IBM Content Navigator is deployed, or into a different web server as shown in Figure 9-1. This simulation type is appropriate for the approaches described in these sections:

- 9.4, “Integrating Content Navigator with URL API” on page 308
- 9.5, “Integrating Content Navigator with a specific feature” on page 310
- 9.6, “Integrating Content Navigator with a specific layout” on page 313

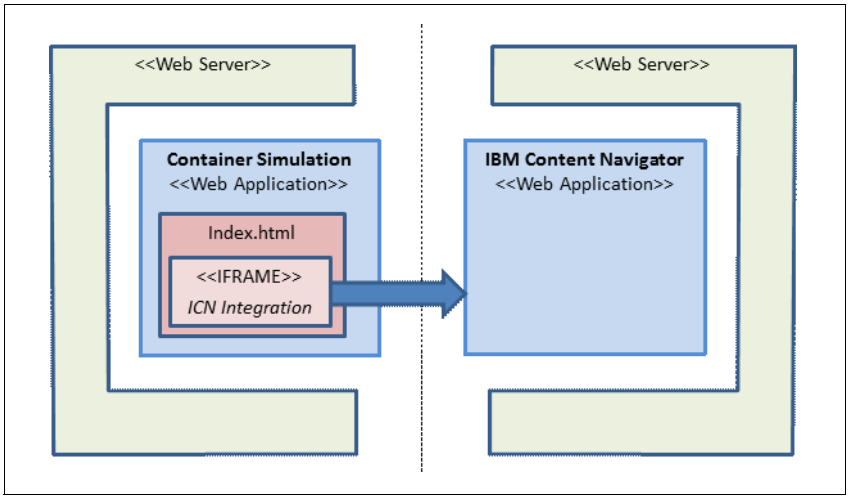



*Figure 9-1 Emulation of an unbound integration of IBM Content Navigator*

Figure 9-2 on page 304 illustrates a bound integration. As you see the code of the IBM Content Navigator integration goes directly into the `Index.html` page, which this time emulates a page of the external application. Bound integration is possible only with the approach described in 9.7, “Integrating specific widgets of Content Navigator” on page 324, because it is the only one in which you can manually set up the IBM Content Navigator widgets directly in the page of the external application.

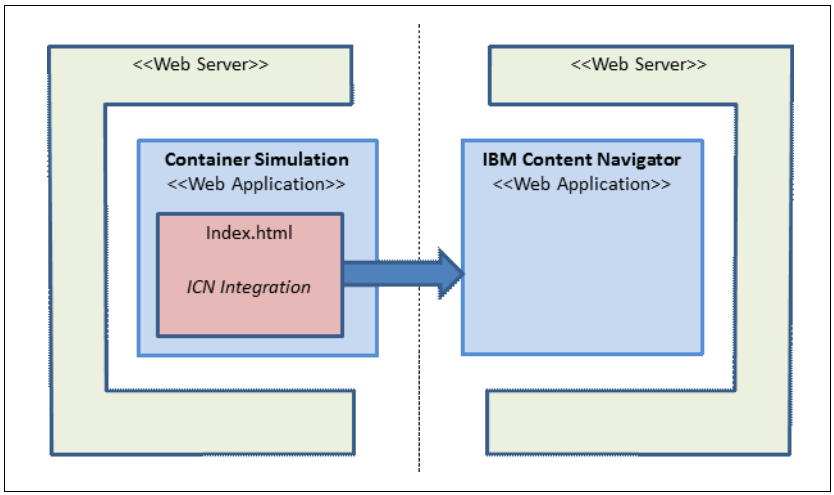



*Figure 9-2 Emulation of a bound integration of IBM Content Navigator*

If you deploy the Container Simulation application in a remote server as illustrated, you can already determine whether same-origin policy violations are issued and you can try to solve them. These kinds of challenges are described in 9.8, “Integrating the externalized widgets (Step 5)” on page 344.

With these two simulation types, we can now test the various approaches before we integrate the externalized IBM Content Navigator into a specific environment in the remainder of this chapter. So the whole first part is independent of a specific target system.

In the IDE of your choice, create a new web project and add a welcome file, named `index.html`. Figure 9-3 shows the basic web project structure, after adding the `index.html` file.

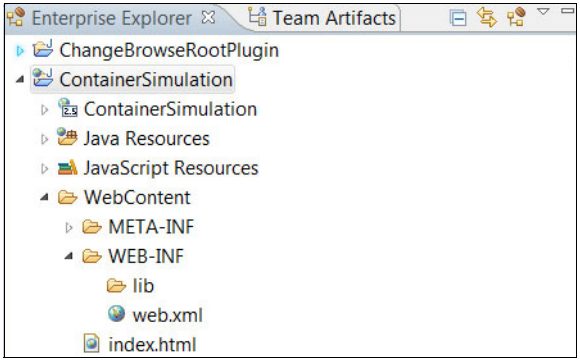



*Figure 9-3 Project structure of the container simulation web project*

A Java web application has a deployment description file `web.xml` that is in the the `WEB-INF` directory. For our purpose, it can be basic and has only one main entry that registers `index.html` as welcome file. Example 9-1 shows the `web.xml` of this web project.

*Example 9-1 web.xml of Container Simulation project*

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <display-name>ContainerSimulation</display-name>
   <welcome-file-list>
      <welcome-file>index.html</welcome-file>
   </welcome-file-list>
</web-app>
```

The `index.html` welcome page is structured with three DIV tags. The first and the last DIV tags represent content of the external application; the middle DIV contains content of the externalized IBM Content Navigator. See Example 9-2.

In this file, the integration is done with an iFrame element, so this is an unbound integration. The URL in the `src` attribute will be adapted in the different approaches of the next sections. For a first test, adapt the URL to point to your IBM Content Navigator deployment.

For a simulation of the bound integration, in 9.8.4, “Outline of a bound integration” on page 349, we show how the externalized IBM Content Navigator widget is initialized directly in the middle DIV tag, which simulates a direct integration into the external application.

*Example 9-2 index.html of Container Simulation project*

```
<!DOCTYPE html >
<html>
<head>
   <title>Container for IBM Content Navigator Integration</title>
   <style>
      @import "ContainerSimulation.css";
   </style>
</head>
<body>
<div class="externalAppContent">
   Here is some information from the Container
</div>
<div id="icnIntegration">
   <iframe class="icnIntegrationFrame"
```

```
      src="http://<webserver>:<port>/navigator"
      name="IBM Content Navigator" seamless>
   </iframe>
</div>
<div class="externalAppContent">
   Here is another piece of information from the Container
</div>
</body>
</html>
```

For providing initial CSS styles, add a new `ContainerSimulation.css` file to the `WebContent` folder and provide an implementation. See Example 9-3 for a starting point.

*Example 9-3 Implementation of* `ContainerSimulation.css`

```
iframe.icnIntegrationFrame{
   width: 800px;
   height: 600px;
   overflow: auto;
}
.externalAppContent{
   background-color:#ddddff;
   height: 50px;
   text-align: center;
   font-size: 24px;
}
#icnIntegration{
   background-color:#ddddff;
   margin: 0 auto;
    text-align: center;
}
```

Compress your web application to a web archive by using the export function of your IDE. The result is an archive file; in our case it is `ContainerSimulation.war`. Deploy this web archive to the web server where IBM Content Navigator is deployed. For the remainder of this section, we assume that you set the web context to `ContainerSimulation`.

Invoke your Simulation Container web application by launching a web browser with URL such as in the following example:

```
http://<web server>:<port>/ContainerSimulation
```

After you log in, the page shown in Figure 9-4 opens.

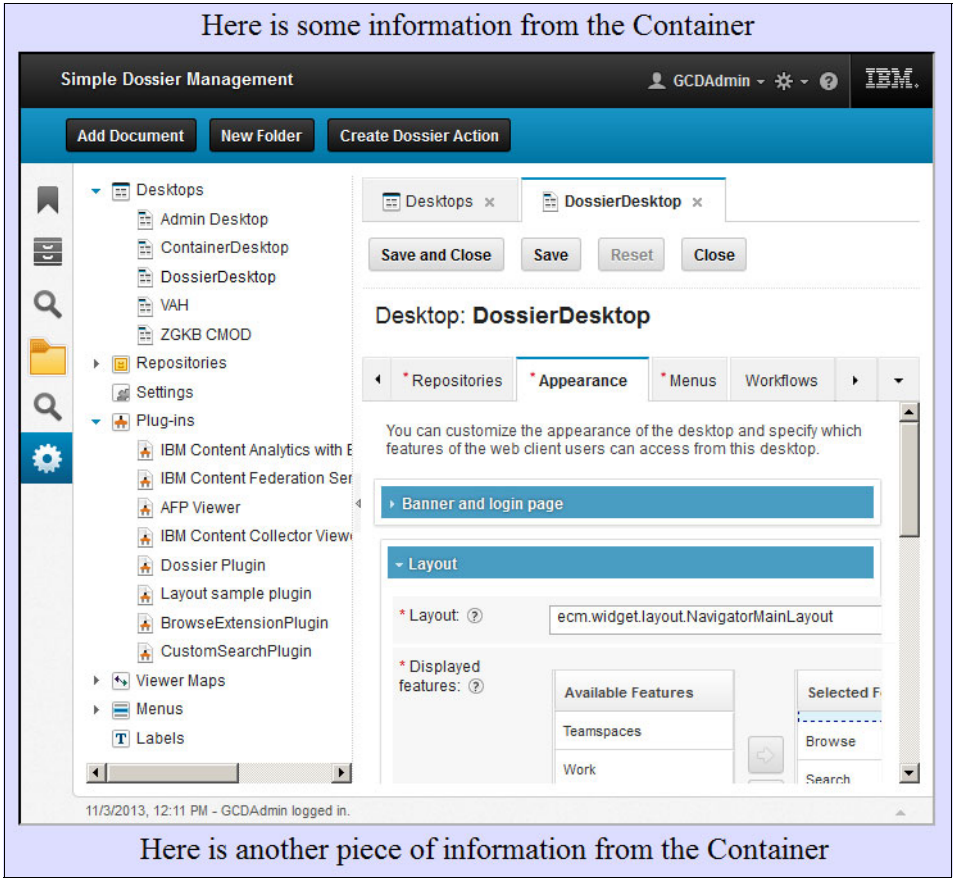



*Figure 9-4 Deployed Simulation Container web application with unbound integration*

Now that we have a web project to emulate IBM Content Navigator integration of the next part of this chapter, we can start with the various approaches. This mock container will be substituted by a real target system.

# 9.4 Integrating Content Navigator with URL API

The first way of an unbound integration of IBM Content Navigator into another application is to invoke its URL and let IBM Content Navigator appear “as-is.” The simplest way is to invoke only the start page of IBM Content Navigator without additional parameters, which was already done in the previous section.

As introduced in 1.4.1, “URL API” on page 23, IBM Content Navigator exposes a URL API that provides some control of what to show to the user, through deep linking.

You can mainly specify the following elements:

- Desktop
- Feature
- Folder
- Document

As a sample for this approach, we show the content of a specific folder.

If you want to display the content of a specific folder, you must invoke a different start page as shown in Example 9-4. The `docid` parameter specifies the unique identifier of the folder. For FileNet P8, this is the property ID, which is a Globally Unique Identifier (GUID) of the object.

*Example 9-4 Display the content of a specific folder*

```
http://server:port/navigator/bookmark.jsp?desktop=DossierDesktop&featur
e=browsePane&docid=6DEFC57F-6E89-4B4B-BD04-D4B5DC5E02E8
```

First, you can verify the constructed URL by copying it directly into the browser. The result is similar to Figure 9-5 on page 309.

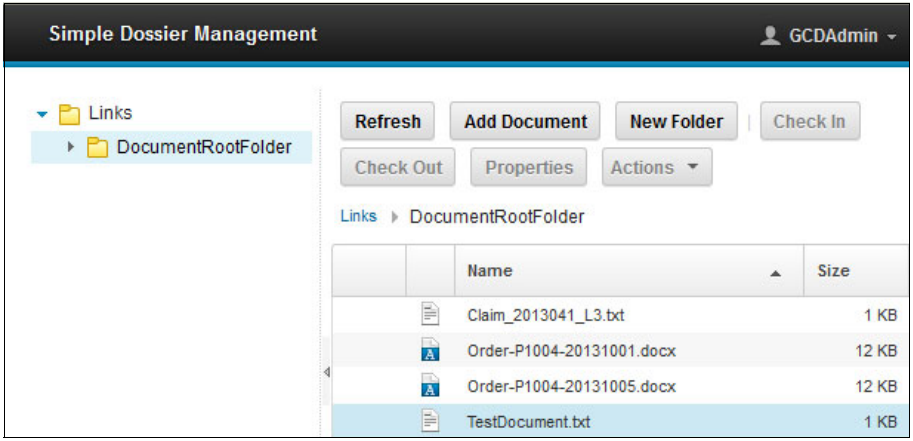



*Figure 9-5 Display the content of a folder with IBM Content Navigator URL API*

Next, you can also try the integration with the Simulation Container application that was created in 9.3.2, “Simulation of IBM Content Navigator integration part 1” on page 303. To do that, edit the src attribute of the `iframe` tag in `index.html` (shown in Example 9-5). If you empty your browser cache and restart your Container Simulation application in the browser, you should find the folder content in the iFrame and that is in the simulated external application.

*Example 9-5 index.html of Container Simulation project with deep linking*

```
<div id="icnIntegration">
   <iframe class="icnIntegrationFrame"
src="http://server:port/navigator/bookmark.jsp?desktop=DossierDesktop&f
eature=browsePane&docid=6DEFC57F-6E89-4B4B-BD04-D4B5DC5E02E8"
      name="IBM Content Navigator" seamless="true">
   </iframe>
</div>
```

The control of IBM Content Navigator is restricted to the URL API. If this is not enough, you might consider enhancing the URL API. We do not provide details for this enhancement, but can give you some guidance. To better understand, see 9.7.1, “Initialization phase of Content Navigator” on page 325.

- When you start IBM Content Navigator using URL API and show the content of a folder, specify a different start page: instead of the `launch.jsp` page, specify `bookmark.jsp`.
- The `bookmark.jsp` file basically calls `browserbookmark.jsp` for non-mobile access.

- The `browserbookmark.jsp` file launches the specified component (for example, a folder's content) of IBM Content Navigator with the BookmarkPane widget.
- The `ecm.widget.BookmarkPane.js` widget overrides the standard basic widget for IBM Content Navigator, which is `ecm.widget.DesktopPane`. The following code snippet is of `BookmarkPane.js`:

  ```
  declare("ecm.widget.BookmarkPane", [DesktopPane,LoggerMixin,
  MessagesMixin ], {...});
  ```

  In the desktopLoaded function, the layout is set to BookmarkLayout and is started. That means the IBM Content navigator is not started with the standard Layout, which is `NavigatorMainLayout`, but with BookmarkLayout.
- The `ecm.widget.layout.BookmarkLayout.js` widget handles the additional URL parameters like docid.

To enhance the URL API functionality, you can provide these items:

- Provide your own start page, such as `mybookmark.jsp`.
- The `mybookmark.jsp` can launch a myBookmarkPane widget, which extends `BookmarkPane.js`.
- The `myBookmarkPane.js` can specify a layout that extends the BookmarkLayout and provides the additional functionality you need.

This works for similar extensions as provided in the URL API of IBM Content Navigator. If you want more control or must direct IBM Content Navigator to a specific widget, consider using a different approach, as described in the next sections.

## 9.5 Integrating Content Navigator with a specific feature

The next approach to integrate parts of IBM Content Navigator into an external application is to leverage a single feature of IBM Content Navigator.

To open a specific feature and desktop in IBM Content Navigator, you can use the IBM Content Navigator URL API as explained in the previous section. The URL is constructed like the one in Example 9-6.

*Example 9-6 Display a specific desktop and feature of IBM Content Navigator*

```
http://server:port/navigator/?desktop=<desktopid>&feature=<featureid>&s
ideChrome=0
```

The standard feature IDs are as follows:

- browsePane: for the standard browse feature
- searchPane: for the standard search feature
- favorites: for the standard favorite feature
- manageTeamspaces: for the standard teamspace feature
- workPane: for the standard work feature

Be sure the specified feature is properly configured and assigned to the given desktop.

With the additional URL parameter, sideChrome, you can also hide the bar for features on the left side and the top banner for a more seamless integration.

When the feature is a standard feature of IBM Content Navigator, this integration approach has the same level as in 9.4, “Integrating Content Navigator with URL API” on page 308. But developing a custom feature provided through a plug-in gives you a different level of freedom. When you provide a custom feature, you can start from an empty page and add exactly the widgets you want to expose to the external application. And if you set the sideChrome parameter to 0 (zero), you have a seamless integration because only the feature is rendered without anything additional from the IBM Content Navigator standard application, such as a banner on the top or a feature bar on the left.

Several sections in this book describe how to create your own feature, so we do not explain it here. You can experiment with any of the feature plug-ins you created in other chapters, for example the dossier feature plug-in from 4.5, “Open dossier view in its own feature” on page 156.

To give you an idea of how this approach might look, we invoke the Work feature with the sideChrome parameter. We invoked the following constructed URL:

```
http://<server>:<port>/navigator?desktop=<desktopid>&feature=workPane&s
ideChrome=0
```

You can invoke the URL directly as shown in Figure 9-6 on page 312. It shows only the application spaces.

If you open it, you can select an in-basket, which lists work items, as illustrated in Figure 9-7.

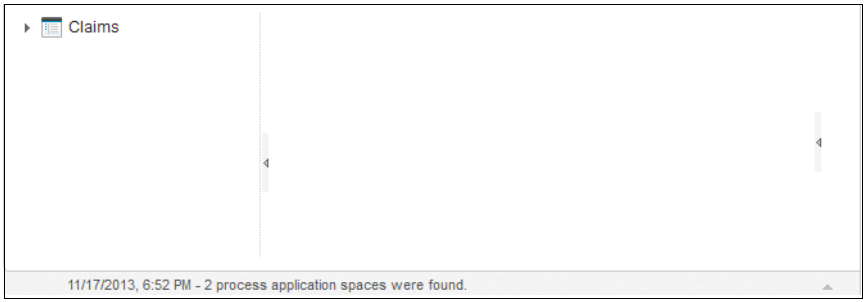



*Figure 9-6 Externalize Work feature with sideChrome parameter*

But, opening the application space and selecting the in-basket are two manual steps, which must be done by the user. If you want to directly open a specific inbox, you must either extend the URL API of IBM Content Navigator as described in the previous section or develop your own work feature that can read additional URL parameters (for example applicationSpace and inbasket) and directly open the work item list. Your custom work feature is then deployed as a plug-in, which implements the feature extension point.

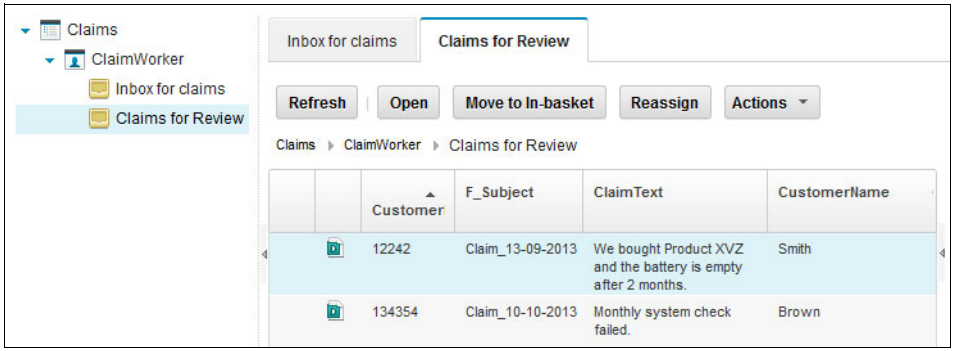



*Figure 9-7 Externalize Work feature with selected in-basket*

This is again a sample for an unbound integration. After integrating it into the Container Simulation application by setting the `src` attribute of the `IFRAME` tag (as shown in Example 9-5 on page 309), it is similar to Figure 9-8 on page 313. This is after manually selecting an application space and an in-basket, and maximizing the work item list with the small triangular grip located between the in-basket and the work item list area.

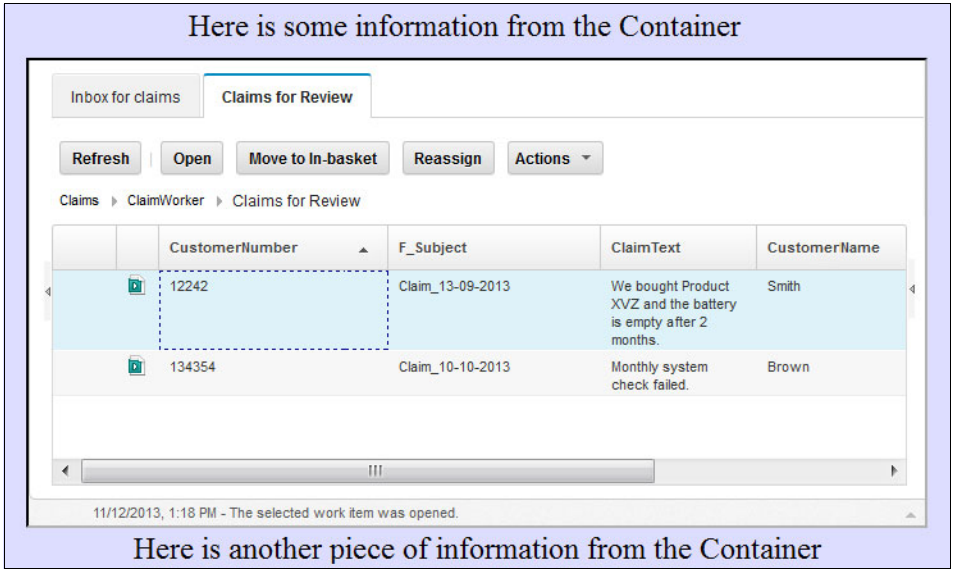



*Figure 9-8 Integrating the work pane feature into the Simulation Container*

## 9.6 Integrating Content Navigator with a specific layout

In this section, we implement a new layout for the layout extension point of the IBM Content Navigator plug-in. With this approach, we still use the IBM Content Navigator standard application but we gain total freedom regarding which widgets to display and how to arrange them. In the sample implementation, we expose the ContentList widget. No folder tree for navigation should be provided.

The content list widget is explained in 5.2, “Viewing results in ContentList widget” on page 174. It will be helpful if you have familiarized yourself with this widget before starting this section.

### 9.6.1 Setting up a new plug-in project

Create a new project in your IDE; use the parameters listed in Table 9-1 on page 314. We do not provide further instructions for this step (it is explained in 3.3, “Plug-in development” on page 81).

*Table 9-1 Parameters for IBM Content Navigator plug-in project*

| Parameter | Value |
|---|---|
| Descriptive name | `Layout Plugin` |
| Java package | `com.ibm.ecm.extension` |
| Class name | `LayoutPlugin` |

Your project will be similar to Figure 9-9.

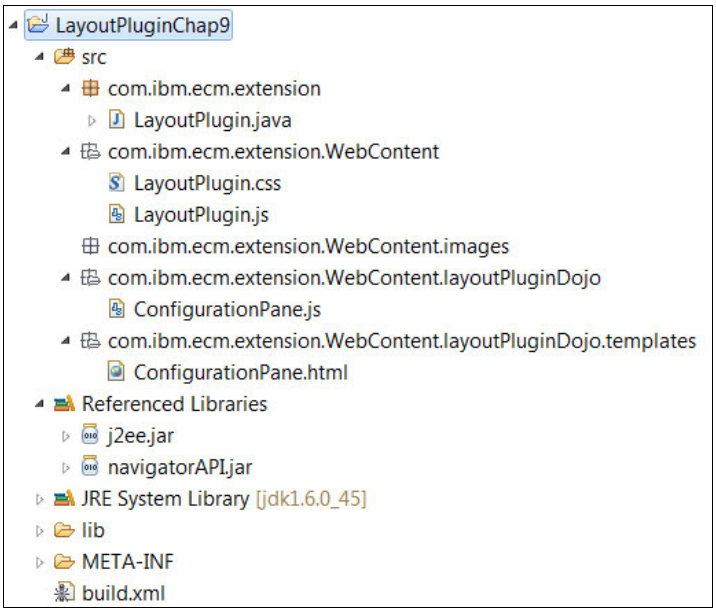



*Figure 9-9 Structure of Layout Plugin project*

The next step is to add a new layout to the plug-in.

## 9.6.2 Adding the layout

This section provides an implementation of an IBM Content Navigator layout extension point. To add a new layout to the plug-in, complete these steps:

1. Create a specific `com.ibm.ecm.extension.PluginLayout` Java class and implement its methods to add the functionality you need.
2. Hook it to the base Plugin class by adding your PluginLayout class, overriding the getLayouts() method.

Both tasks can be done with the New Layout wizard of the Eclipse Plugin for IBM Content Navigator development. See 3.3.2, “Creating plug-in extension” on page 85. Use ContentListLayout as class name. This creates a skeleton of the class with basic implementations for each of the abstract methods, see Example 9-7.

*Example 9-7 Implementation of ContentListLayout.class*

```
public  class ContentListLayout extends PluginLayout {
   public  String getId() {
      return "ContentListLayout";
   }
   public  String getName(Locale locale) {
      return "ContentListLayout";
   }
   public boolean areFeaturesConfigurable() {
      return false;
   }
   public  String getLayoutClass() {
      return "layoutPluginDojo.ContentListLayout";
   }
}
```

Keep the default implementation except for the `areFeaturesConfigurable` method. We change the default value to `false` because we do not want to support the concept of IBM Content Navigator features, we only render the content list widget.

Example 9-8 shows the corresponding `ContentListLayout.js` file, which is the layout widget.

*Example 9-8 Implementation of ContentListLayout.js*

```
define([
      "dojo/_base/declare",
      "dijit/_TemplatedMixin",
      "dijit/_WidgetsInTemplateMixin",
      "dijit/layout/StackContainer",
      "dijit/layout/BorderContainer",
      "dijit/layout/ContentPane",
      "ecm/widget/layout/BaseLayout",
      "ecm/widget/Banner",
      "ecm/widget/LoginPane",
      "ecm/model/Feature",
      "dojo/text!./templates/ContentListLayout.html"
   ],
```

```
   function(declare, _TemplatedMixin, _WidgetsInTemplateMixin,
StackContainer, BorderContainer, ContentPane, BaseLayout, Banner,
LoginPane, Feature , template) {

   return declare("layoutPluginDojo.ContentListLayout", [ BaseLayout,
_TemplatedMixin, _WidgetsInTemplateMixin], {
      templateString: template,
      widgetsInTemplate: true,

      getAvailableFeatures: function() {
         return [
            new Feature({
               id: "favorites",
               name: "favorites",
               featureClass: "ecm.widget.layout.FavoritesPane",
            })
         ];
      },
   });
});
```

We suggest implementing the `getAvailableFeatures` function, even if you do not use any feature. Otherwise the Appearance tab on the Administration Desktop will not correctly render the "Displayed features:" section.

The code defines the layout with the common Dojo template mechanism. The corresponding template file is defined:

`dojo/text!./templates/ContentListLayout.html`

The wizard has generated the layout with a StackContainer similar to the `ecm/widget/layout/templates/MainLayout.html`; see Example 9-9 on page 317. This is a good starting point if you want to build a complex application. We just want to place our content list widget here. So basically we can delete everything and start with one `DIV`. The reason why we keep the StackContainer is that IBM Content Navigator tries to set the focus on `mainStackContainer`, defined in the `data-dojo-attach-point`. And, if we delete it, we get an error. So we keep the StackContainer, and delete just the top banner definition.

The only item we put in the content area of the StackContainer now is a marker to verify that everything is working correctly when we deploy the plug-in.

*Example 9-9 ContentListLayout.html*

```
<div class="ecmLayout">
   <div data-dojo-type="dijit/layout/BorderContainer"
data-dojo-attach-point="mainContainer" data-dojo-props="gutters:false"
class="contentPane">
      <div data-dojo-type="dijit/layout/StackContainer"
          data-dojo-attach-point="mainStackContainer"
          data-dojo-props="region:'center'"
          class="stackContainer">
         <div data-dojo-type="ecm/widget/LoginPane"
data-dojo-attach-point="loginPane" id="${id}_LoginPane"></div>

         <div data-dojo-type="dijit/layout/BorderContainer"
data-dojo-attach-point="mainPane" data-dojo-props="gutters:false">
            <!-- Add your main application layout here -->
            Our content widget will go here.
         </div>
      </div>
   </div>
</div>
```

As we take an iterative methodology, we already deploy our new plug-in, see Figure 9-10 on page 318. At this time, we do not build a JAR file for the plug-in, but configure the Class file path. This allows us to directly enhance the code without having to rebuild a JAR file each time we want to see the effects of our implementation.

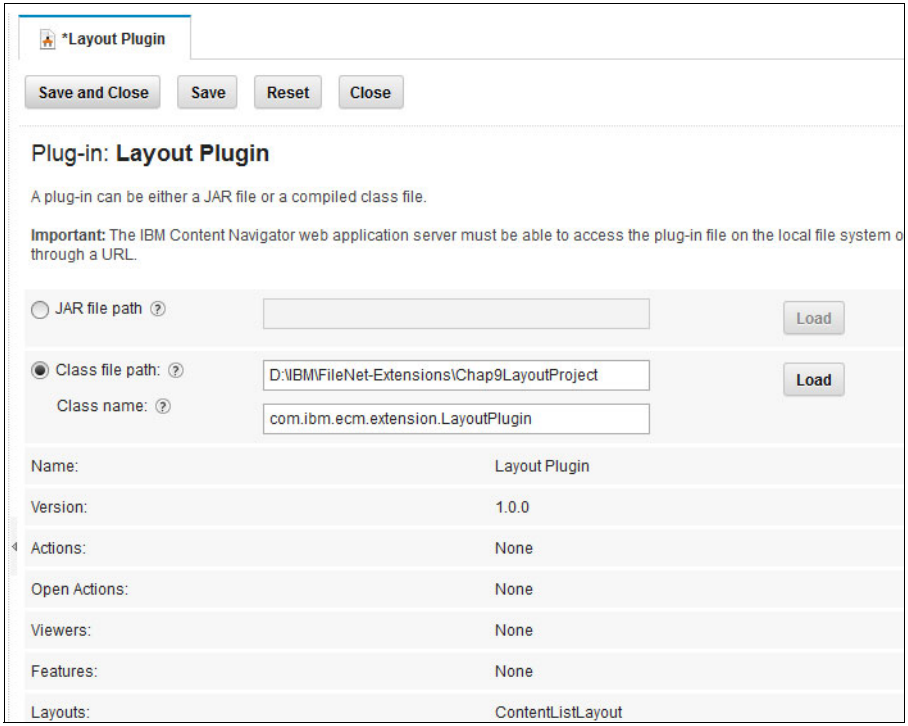



*Figure 9-10 Configure LayoutPlugin*

To know how the new layout looks, we must assign it to a desktop. Complete the following steps in the Administration view feature:

1. You can assign the layout to the desktop of your choice. From the Administration view, create a new desktop by selecting **Desktops** → **New Desktop** and enter `ContentListDesktop` as the name of the desktop with the following configurations:
   - General tab:
     - Name and ID: ContentListDesktop
     - Authentication: Select the repository of the folder that you want to list in the ContentList widget.
   - Repositories tab: Assign this repository again.
   - Appearance tab (see Figure 9-11 on page 319):
     - Layout: Assign the ContentListLayout in the drop-down box.
     - Displayed features: Basically we do not need a feature. But, because specifying a default feature is mandatory, we must provide at least one

feature, so we choose the **Favorites** feature, which we previously specified in `ContentListLayout.js`.

- Default feature: Favorites.
- Default repository: Disable this because the favorites feature does not need to be initialized with a repository.

2. Save the desktop. If the Save button is not enabled, update the state of the Save button as follows:
   a. Select **Favorites** in the Selected Features area.
   b. Move Favorites to the Available Features area.
   c. Move Favorites it back again to the Selected Features area.

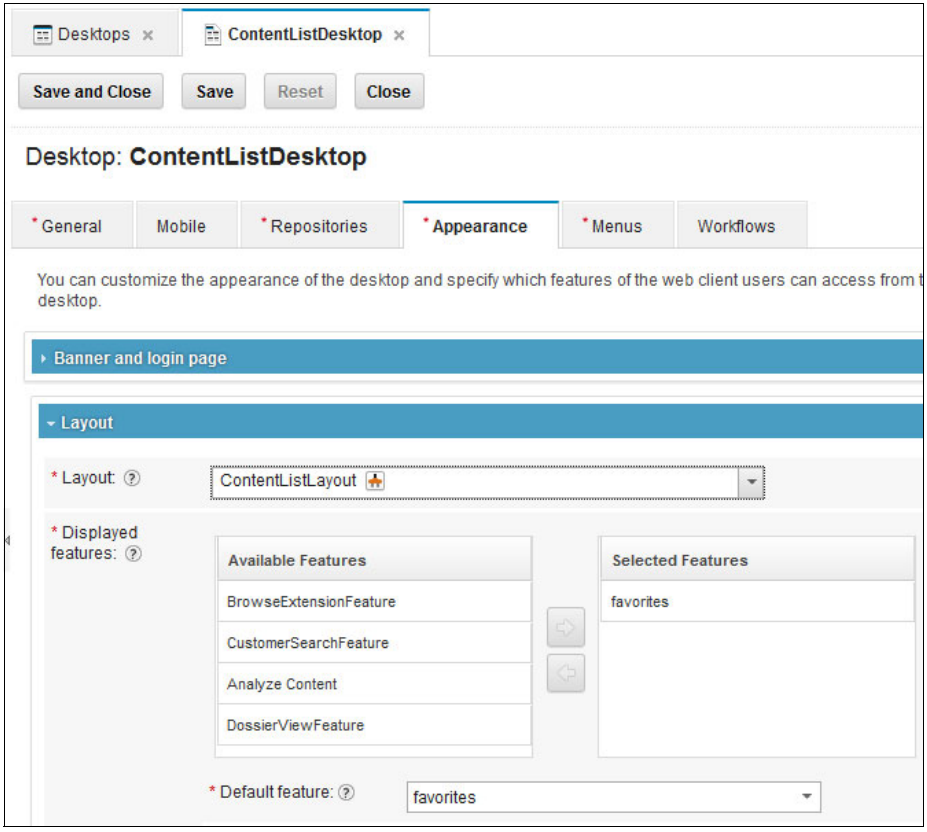



*Figure 9-11 Assigning the ContentListLayout to the new desktop*

When you start IBM Content Navigator with this desktop, such as with the URL in the following example, you see an empty page, filled only with the `Our content widget will go here` marker:

```
http://server:port/navigator?desktop=ContentListDesktop
```

### 9.6.3 Adding the ContentList widget to the layout

In the template HTML file, we add the ContentList widget; see Example 9-10.

*Example 9-10 Adding ContentList to ContentListLayout.html*

```
<div class="ecmLayout">
   <div data-dojo-type="dijit/layout/BorderContainer"
data-dojo-attach-point="mainContainer" data-dojo-props="gutters:false"
class="contentPane">
       <div data-dojo-type="dijit/layout/StackContainer"
            data-dojo-attach-point="mainStackContainer"
            data-dojo-props="region:'center'"
            class="stackContainer">
           <div data-dojo-type="ecm/widget/LoginPane"
data-dojo-attach-point="loginPane" id="${id}_LoginPane"></div>
           <div data-dojo-type="dijit/layout/BorderContainer"
data-dojo-attach-point="mainPane" data-dojo-props="gutters:false"><!-- Add your
main application layout here -->
           <div data-dojo-attach-point="contentList"
data-dojo-type="ecm/widget/listView/ContentList" data-dojo-props="region:'center'">
           </div>
       </div>
   </div>
</div>
```

Dojo framework provides a reference to the `ecm/widget/listView/ContentList` widget with the `this.contentList` property (specified in the data-dojo-attach-point attribute) in the `ContentListLayout.js` file. Example 9-11 shows the postCreate function that we override to initialize our ContentList widget. In the Dijit lifecycle, this method is often a good place to do initialization tasks.

We set ContentListModules and ContentListGridModules, invoke the displayRoot method, which retrieves the content of the root folder of the repository, and initialize ContentList widget with the ResultSet.

*Example 9-11 Initialization of ContentList in postCreate method of ContentListLayout.js*

```
postCreate : function() {
   this.inherited(arguments);
   this.contentList.setContentListModules(
      this.getContentListModules());
   this.contentList.setGridExtensionModules(
      this.getContentListGridModules());
   this.displayRoot();
},
```

Example 9-12 on page 321 shows the helper methods.

The first two methods to configure the content list modules and the modules themselves are explained in 6.5, “Configuring more modules to the ContentList widget” on page 225.

The `displayRoot` method gets the default repository from the desktop. The verification that the repository is defined is necessary because the layout is also loaded for configuration purposes in the administrative feature; at that time, the desktop is not loaded and therefore no repository is defined. From the repository, we retrieve the root item and retrieve, in its callback function, its folder content. The retrieveFolderContents method also has a callback function, which has the folder content filled into the resultSet parameter. Finally, the contentList is initialized with this resultSet.

*Example 9-12 Helper methods of postCreate in ContentListLayout.js*

```
getContentListGridModules : function() {
   var array = [];
   array.push(DndRowMoveCopy);
   array.push(DndFromDesktopAddDoc);
   array.push(RowContextMenu);
   return array;
},

getContentListModules : function() {
   var viewModules = [];
   viewModules.push(ViewDetail);
   viewModules.push(ViewMagazine);
   var array = [];
   array.push({
      moduleClass : Bar,
      top : [ [ [ {
         moduleClass : Toolbar
      }, {
         moduleClasses : viewModules,
         "className" : "BarViewModules"
      } ] ], [ [ {
         moduleClass : Breadcrumb
      } ] ] ]
   });
   array.push(DocInfo);
   return array;
},

displayRoot : function() {
   var repository = ecm.model.desktop.getDefaultRepository();
   if (repository) {
```

```
      repository.retrieveItem("/",
         lang.hitch(this, function(rootItem) {
            rootItem.retrieveFolderContents(false,
               lang.hitch(this, function(resultSet) {
                  this.contentList.setResultSet(resultSet, rootItem);
            }));
      }));
   }
}
```

Next, we must define the Dojo modules we need in the helper methods. They are highlighted in the code (Example 9-13); at the beginning of `ContentListLayout.js`, we add the declarations.

*Example 9-13 Dependent modules for ContentListLayout.js*

```
define([
   "dojo/_base/declare",
   "dijit/_TemplatedMixin",
   "dijit/_WidgetsInTemplateMixin",
   "dijit/layout/StackContainer",
   "dijit/layout/BorderContainer",
   "dijit/layout/ContentPane",
   "ecm/widget/layout/BaseLayout",
   "ecm/model/Feature",
   "ecm/widget/listView/gridModules/RowContextMenu",
   "ecm/widget/listView/modules/Breadcrumb",
   "ecm/widget/listView/modules/Bar",
   "ecm/widget/listView/modules/Toolbar",
   "ecm/widget/listView/modules/DocInfo",
   "ecm/widget/listView/gridModules/DndRowMoveCopy",
   "ecm/widget/listView/gridModules/DndFromDesktopAddDoc",
   "ecm/widget/listView/modules/ViewDetail",
   "ecm/widget/listView/modules/ViewMagazine",
   "ecm/widget/listView/ContentList",
   "ecm/model/Desktop",
   "dojo/_base/lang",
   "dojo/text!./templates/ContentListLayout.html"
   ],
   function(declare, _TemplatedMixin, _WidgetsInTemplateMixin,
StackContainer, BorderContainer, ContentPane, BaseLayout, Feature,
RowContextMenu,Breadcrumb, Bar, Toolbar, DocInfo, DndRowMoveCopy,
DndFromDesktopAddDoc, ViewDetail, ViewMagazine, ContentList,
Desktop,lang, template) {
```

If you load and save your plug-in configuration in the administrative pane of IBM Content Navigator, you see the ContentList widget when you invoke the IBM Navigator with the following URL:

```
http://server:port/navigator?desktop=ContentListDesktop
```

The integration is done with URL invocation so it is still an unbound integration. So the integrating into the Container Simulation application is done through the `iframe src` attribute according to Example 9-14.

*Example 9-14 index.html after integrating the Navigator with ContentList widget*

```
<div id="icnIntegration">
   <iframe class="icnIntegrationFrame"
      src="http://w2008r2p85:9080/navigator/?desktop=ContentListDesktop
      name="IBM Content Navigator" seamless="true">
   </iframe>
</div>
```

After invoking the Container Simulation application in the browser, the results are similar to Figure 9-12.

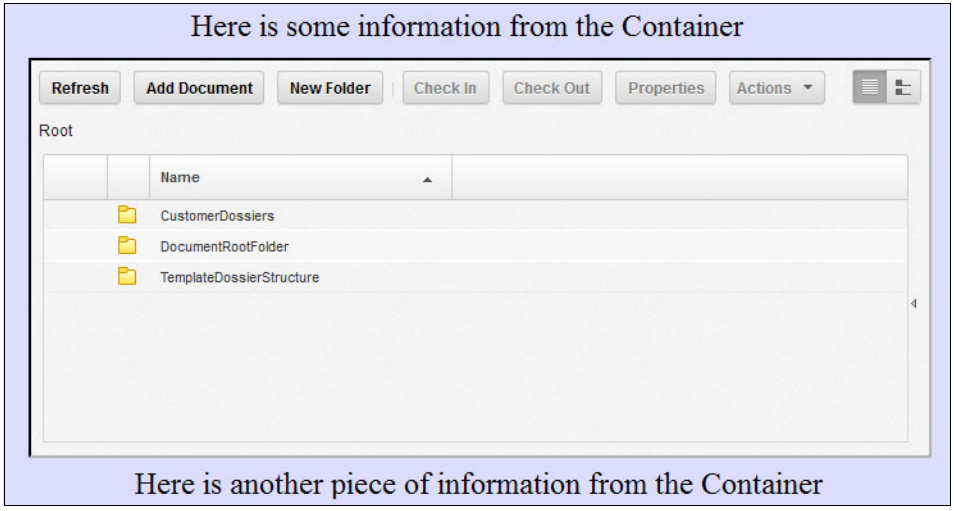



*Figure 9-12 Integration of ContentList widget with custom layout into the container*

## 9.7 Integrating specific widgets of Content Navigator

In this section, we show the most flexible approach: we externalize only the IBM Content Navigator widgets that we need, with as little initialization as needed.

The main difference to the previous approaches is that we do not invoke the normal start page of IBM Content Navigator, which will initialize the specified desktop with full functionality and features. Rather, we set up only the specific pieces of the IBM Content Navigator that we need for the integration.

Because the architecture of IBM Content Navigator is built as a flexible and extensible framework, you can use many of its components. The IBM Content Navigator JavaScript model and the IBM Content Navigator visual widget library provide the building blocks to create your own user interface.

The model is used to ease the access to midtier services for the client-side components and to have one common place to hold the data for the visual widgets which prevents that every widget holds the same date. In general, the model consists of classes that provide non-visual business logic and data that is shared across multiple visual widgets.

IBM Content Navigator user interface is built on visual widgets, arranged in a layout and wired together through events. The model and visual widgets publish and listen to events that can be used to trigger actions and changes to visual components.

In this section, we initialize IBM Content Navigator widgets in a single web page. With this approach we can realize both unbound and bound integration:

- For an unbound integration we can deploy this page either in the IBM Content Navigator deployment or in its own web application inside the application server of IBM Content Navigator. In any case, we invoke the URL of this web page in the external application; if the external application is a web application, the IBM Content Navigator widgets will most likely run inside an iFrame.

  You can also use this web page as stand-alone application. The steps of implementing that page are nearly the same as if you were building a custom application that uses the IBM Content Navigator widgets.

- For a bound integration the implementation of such a single web page can be just a starting point. For the real integration you must transfer that page into the external application or container. For example, for a container like Microsoft SharePoint, you must create a web part which runs the IBM Content Navigator widgets and in a Portlet container like IBM WebSphere Portal server you must create a portlet which leverages the IBM Content Navigator widgets.

Complete the following tasks to set up an application that uses IBM Content Navigator widgets:

1. Initialize Dojo and IBM Content Navigator libraries
2. Select the appropriate widgets from the visual widget library.
3. Select and initialize the necessary model classes
4. Wire the widgets together through event registration.
5. Integrate the externalized widget in the external application

The last step is different for the two type of integrations:

- For an unbound integration, the externalized widget has to be deployed and invoked by the external application.
- For a bound integration, this is where you set up and run the externalized widget inside the container of the external application.

To be able to set up the IBM Content Navigator widget, understanding how IBM Content Navigator is initialized is helpful. This is why we help first examine the initialization phase of IBM Content Navigator.

Next, follow the steps and implement the externalization of an IBM Content Navigator widget. The sample code does the following tasks:

1. Accesses the desktop and the configured default repository.
2. Presents a login dialog where users enter their credentials.
3. Loads a specific folder or the root folder in the ContentList widget.

## 9.7.1 Initialization phase of Content Navigator

To determine how to initialize the widget that you want to externalize, understand the initialization phase of IBM Content Navigator web application. For an introduction, see 1.3.2, “Communication flows” on page 20; then, return here for details.

**Notes:**

- This initialization process might not be exactly the same for each release. If you understand the basic methodology or strategy of how IBM Content Navigator widgets must be initialized, then in future versions of IBM Content Navigator, you can take the same approach, even if some details differ slightly, in principle the process will be the same.
- Be aware that this explanation just gives an overview and does skip a lot of details.

Figure 9-13 outlines the initialization process of IBM Content Navigator.

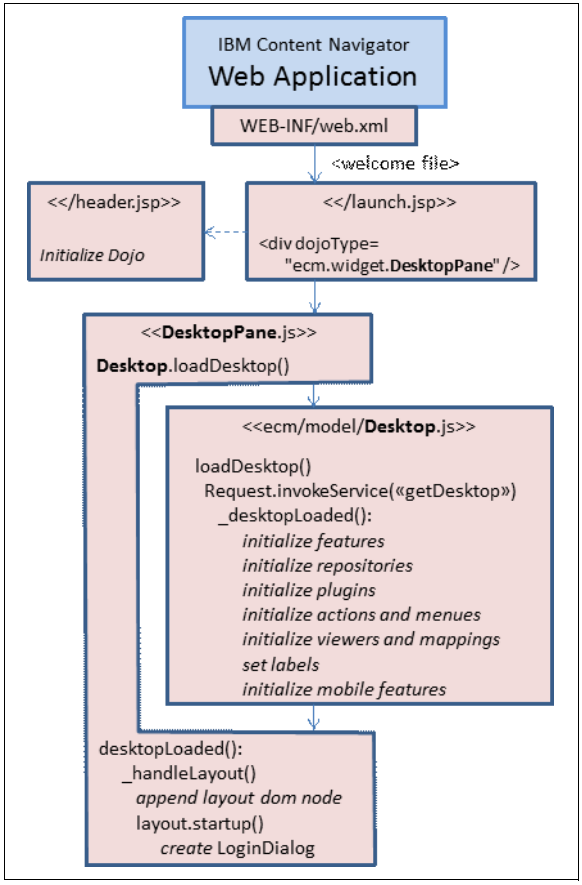



*Figure 9-13 Initialization of IBM Content Navigator*

Consider the following information about Figure 9-13:

- `web.xml`: This deployment descriptor is in the `WEB-INF` directory; every Java web application has a deployment descriptor. The `<welcome file>` tag defines the start page, which is `launch.jsp`
- `launch.jsp`:
  - The included `header.jsp` header file reads a parameter for the application, such as locale, loads the CSS files, and loads and initializes the Dojo libraries.
  - Provides the root `<DIV>` tag, where the whole DOM tree of IBM Content Navigator is built. IBM Content Navigator is implemented as `DesktopPane` widget:

```
<div dojoType="ecm.widget.DesktopPane" id="ECMWebUI"
style="width: 100%; height: 100%"
browserLocale="<%=htmlLocale%>"></div>
```

- `DesktopPane.js` (1 of 2): Defines the IBM Content Navigator widget.

  postCreate(): A specific desktop of available desktop configurations is loaded:

  ```
  Desktop.loadDesktop(this.desktopId, lang.hitch(this,
  "desktopLoaded"));
  ```

- `Desktop.js`: Defines the model of a desktop for IBM Content Navigator.
  - loadDesktop() loads the configuration for the desktop through midtier service from the IBM Content Navigator database:

    ```
    Request.invokeService("getDesktop", null,{...},
    function(response) {
    self._desktopLoaded(response, callback);
    }, false, synchronous);
    ```

  - _desktopLoaded() is invoked after the desktop is loaded. This is the main method for initialization for a lot of key aspects of IBM Content Navigator:
    - Initialize feature models
    - Initialize repository models
    - Initialize plug-in models:
      - Load plug-in CSS files
      - Load plug-in Dojo module
      - Load plug-in script file and execute the script
    - Initialize action and menu models
    - Initialize viewers and viewer mappings
    - Set configured labels
    - Initialize mobile feature models

- `DesktopPane.js` (2 of 2):

  desktopLoaded() is invoked after the desktop model is loaded and other models are initialized. It mainly loads and initializes the layout, which arranges the widgets for the desktop in _handleLayout():

  ```
  this.layout.setFeatures(Desktop.features, Desktop.defaultFeature);
  this.domNode.appendChild(this.layout.domNode);
  this.layout.startup();
  ```

- `BaseLayout.js` (in `ecm.widget.layout package`):

  Each Layout that implements the layout for IBM Content Navigator should have BaseLayout as the parent class.

Among others, postCreate() methods of BaseLayout handles the LoginDialog that is invoked when launching IBM Content Navigator for the first time and when the session is expired.

- `NavigatorMainLayout.js` is based on `MainLayout.js` (in `ecm.widget.layout` package):

  It is the default IBM Content Navigator layout, arranges the available features, and shows up with the default (or specified) feature. In the previous section we substituted this default layout with a custom layout plug-in that rendered only the ContentView widget.

  The details of how the default layout of IBM Content Navigator is implemented is out of scope for this book and is not needed for the purpose of this chapter.

If you want to get an idea of how to set up the models and widgets of IBM Content Navigator, seeing how it is done when IBM Content Navigator initializes itself in the start-up phase can be helpful.

## 9.7.2 Step 1: Initialize Dojo and Content Navigator libraries

Before selecting and integrating a Dojo widget, we must set up an HTML page and do some initialization to enable Dojo and to make the IBM Content Navigator widgets available on that page.

**Note:** For a bound integration into an external application, you probably must do this initializing somewhere in an existing HTML page of the external application.

Here we start with a separate HTML page, which can directly be used for an unbound integration.

We call the page `External_ICN.html`, which we implement in the next sections. The previous section (9.7.1, “Initialization phase of Content Navigator” on page 325) can serve as “compass” when setting up the page.

Example 9-15 illustrates the first step.

*Example 9-15 Implementation of External_ICN.html: Step 1*

```
<!DOCTYPE HTML>
<html lang="en">
   <head>
      <meta charset="utf-8">
      <title>ICN Integration</title>
```

```
      <link rel="stylesheet"
href="/navigator/ecm/widget/resources/dojo.css.jgz" media="screen">
      <link rel="stylesheet"
href="/navigator/ecm/widget/resources/ecm.css.jgz" media="screen">
      <link rel="stylesheet"
href="/navigator/ecm/themes/oneui/dojo.css.jgz" media="screen">
      <link rel="stylesheet"
href="/navigator/ecm/themes/oneui/oneui.css.jgz" media="screen">
      <script>
         dojoConfig = {
            async: true,
            packages:
            [
               {name: "dojo",location: "/navigator/dojo"},
               {name: "ecm",location: "/navigator/ecm"}
            ],
            isDebug: true,
            parseOnLoad: false,
         };
      </script>
      <script src="/navigator/dojo/dojo.js.jgz"></script>
      <script src="/navigator/ecm/ecm.js.jgz"></script>
   </head>
   <body class="ecm oneui">
      <div id="icnWidget">
         Here goes the ICN widget.
      </div>

      <script>
         require(["dojo/parser","dojo/domReady!"],
         function(parser) {
               parser.parse();
         });
      </script>
   </body>
</html>
```

Consider the following information about the `External_ICN.html` file:

- `<!DOCTYPE HTML>`: Defines an HTML5 page.
- The header of the HTML page defines the following items:
  - Four references to Cascading Style Sheets (CSS), which are used by Dojo and IBM Content Navigator widgets
  - `dojoConfig:`

Configuration object for the Dojo framework contains these items:

- `async`: Is set to true because we want to develop a Dojo application with the new loader that uses Asynchronous Module Definition (AMD).
- `packages`: Defines the location of the packages we need.
- isDebug: Sets the debugging mode.
- `parseOnLoad`: Is set to false because we want full control and to explicitly parse the Dojo elements.

– Sources of Dojo and IBM Content Navigator JavaScript libraries.

- `<body class="ecm oneui">` defines the CSS styles for the IBM Content Navigator widgets.
- The body of the HTML page defines one `DIV` tag to hold the IBM Content Navigator widget we will externalize.
- The last `<script>` element becomes the core of this page.

  The require() function creates a closure of JavaScript code and provides the Dojo modules needed to execute that code.

  We declare two modules:

  – The `parser` module will be invoked in the function

  – The `domReady!` module is special mechanism to guarantee that the code is invoked only when the whole DOM structure is created. The exclamation point marks this module to be special. It also does not require a corresponding parameter in the function. The `domReady!` module is important because the parser will parse the DOM structure and we are in asynchronous mode.

  In the function, we will provide the actual code that initializes the models and widgets we need for the page. For now, we only set up the Dojo parser, which will parse the declarative Dojo widgets we will add later. Currently the parser has nothing to do.

Although deploying the page is described only in 9.8, “Integrating the externalized widgets (Step 5)” on page 344, we suggest doing it after each step to test the current state and to detect errors as soon as possible.

In short, to test your page, the easiest way is to copy it to your IBM Content Navigator deployment and invoke the page directly in the web browser.

Using IBM WebSphere as an application server, the location will be similar to the following location:

```
...WebSphere/AppServer/profiles/<profileName>/installedApps/<cellName>/
ContentNavigator.ear/navigator.war/External_ICN.html
```

You can start the browser with the following URL:

```
http://<serverName>:<port>/navigator/External_ICN.html
```

A page shows only the following marker text:

```
Here goes the ICN widget.
```

For further details about deploying the page, see 9.8.1, “Deploying the externalized widget (unbound integration)” on page 346.

## 9.7.3 Step 2: Select the appropriate visual widget

The next step is to browse the catalog of available widgets and select the appropriate candidates, from the visual widget library, to satisfy your requirements.

The widgets are grouped by packages, depending on their functionality. For example if you want to add a visual component to your application that shows a tree view of your repository, select **Folder Tree** from the base widgets package in the document and folder widgets group.

A good introduction of the important IBM Content Navigator widgets and an overview for the JavaScript model library is in 1.4.4, “Content Navigator JavaScript API” on page 47. A complete list of available visual widgets is in the IBM Knowledge Center:

http://publib.boulder.ibm.com/infocenter/cmgmt/v8r4m0/topic/com.ibm.developingeuc.doc/eucrf002.htm

In our sample, we again select **ContentList** widget. This package is introduced in 5.2, “Viewing results in ContentList widget” on page 174 and is known from 9.7.2, “Step 1: Initialize Dojo and Content Navigator libraries” on page 328. This makes the various approaches more comparable. And we can focus on doing it without having to introduce another widget. However, the general considerations in the steps are similar if you select another widget. Finally, we make a small extension and make the widget configurable: with a URL parameter you can pass the folder to be opened in the ContentList widget.

Example 9-16 shows the body of the `External_ICN.html` after adding the ContentList widget.

*Example 9-16 Adding the ContentList widget to External_ICN.html: Step 2*

```
<body class="ecm oneui">
   <div id="contentList"
      data-dojo-type="ecm/widget/listView/ContentList"
```

```
data-dojo-props="plugins:'{dnd:true}',isExternalDND:'true',isResultSetS
orted:'true',copyOnly:'true'" role="region" aria-label="Content List">
   </div>
   <script>
      require([ "dojo/parser","dijit/registry",
         "ecm/widget/listView/ContentList","dojo/domReady!"],
      function(parser, registry, ContentList) {
         parser.parse();
         var contentList = registry.byId("contentList");
      });
   </script>
</body>
```

We replaced the top level DIV tag with one that holds the ContentList widget, which is specified with the HTML5 data-dojo-type attribute. Because we use this declarative approach to specify the Dojo widget, now the Dojo parser has work to do: It must read the DOM and parse nodes that have been decorated with special Dojo attributes. In our case, the ContentList widget is instantiated.

**Note:** We do not refer directly to ContentList in the anonymous callback function of require() but nevertheless it is declared in the data-dojo-type attribute in the ContentList widget declaration. Although, the code might work even if you do not declare the widget, because the Dojo parser is capable of auto-requiring modules, the best practice is to explicitly be sure that the parser has loaded all required modules in advance.

All the widgets are registered and a reference can be fetched with the registry.byId() method. We use this technique to get a reference to the instantiation of the ContentList widget, which we will need later for setting the content list modules and the items to show up in the list.

Several properties can be specified to orchestrate the Content List widget:

- plugins="{dnd: true}"

  This property indicates that you want to support "drag and drop" (DND) within the content list. For example, you want to be able to select rows, and drag them into a folder row.

  **Note:** This parameter is ignored if the repository type is OD (IBM Content Manager OnDemand repository).

- isExternalDND

This property supports dragging from the IBM Content Navigator desktop to the content list. The added document dialog box opens if all of the following conditions are true:

- This property value is set to true.
- You drag a file from the desktop to a folder row.
- You have the authority to add documents to the folder.

- isResultSetSorted

  If the property value is set to true, the server returns sorted results.

- copyOnly

  This property value is used for drag and drop and is used when only in the following situation:

  ```
  plugins="{dnd: true}"
  ```

  The word *copy* in this context (copyOnly), does not mean creating a new copy of the item; it rather means creating another reference.

  The property has two values:

  - False (default): If you drag an item over a content list folder, the default drop action is to move the rows into the folder.

    **Ctrl key:** The user can always press Ctrl during a “drag and drop” operation. If the Ctrl key is pressed during the drop, then the move is not done; instead, a new link is created.

  - True: The default action for dropping onto a folder row is to create a new link to this folder.

    **Note:** If the value is set to true, the user cannot move rows.

Invoking the `External_ICN.html` file in a browser now still shows an empty page. This is because the ContentList renders the content list only if an appropriate model is set as resultSet. Nevertheless, we suggest you do this, because if you have any errors you can fix them immediately.

To get the ContentList widget rendered and see that it is well-defined, we provide a small “dummy” implementation for the necessary `ecm/model/ResultSet` model as shown in Example 9-17.

*Example 9-17 Dummy implementation of the ResultSet model in External_ICN.html*

```
<script>
   require([ "dojo/parser","dijit/registry",
      "ecm/widget/listView/ContentList",
      "ecm/model/ContentItem","ecm/model/ResultSet","dojo/domReady!"],
   function( parser,registry, ContentList, ContentItem, ResultSet) {
      parser.parse();
      var contentList = registry.byId("contentList");

      var itemProperties = {
         attributes: { "{NAME}":"DummyFolder"},
         id:"1",
      };
      var contentItem = new ContentItem(itemProperties);

      var resultSetProps = {
         items: [contentItem],
         structure:{
            cells:[[{ field: '{NAME}', name: 'Name', width: '20em'}]],
         },
      };
      var dummyResultSet = new ResultSet(resultSetProps);
      contentList.setResultSet(dummyResultSet);
   });
</script>
```

First, we create a mock ContentItem that we want to be displayed in the ContentList widget. We set only the required properties ID and attributes.

The ResultSet model will get an array of items, which is initialized with the mock item; a structure provides information about how columns should be rendered. The structure object must be structured in a way that it is understandable for the `gridx.Grid` dijit, which is used by the ContentList widget to display the result list.

Finally we initialize the contentList with the constructed dummy ResultSet. Now, we invoke the page again in the browser. The result is similar to Figure 9-14 on page 335.

*Figure 9-14 ContentList widget with dummy ResultSet model*

Because the ContentList widget uses the Dojo `gridx.Grid` widget, it is structured in a similar modular way. At this moment, only the standard modules are loaded for the ContentList. IBM Content Navigator provides many more modules that we will also add into our sample to make it fully functional. See Example 9-18.

*Example 9-18 Defining the modules for ContentList widget in External_ICN.html*

```
require([ "dojo/parser","dijit/registry",
   "ecm/widget/listView/ContentList",
   "ecm/model/ContentItem","ecm/model/ResultSet",
   "ecm/widget/listView/modules/ViewDetail",
   "ecm/widget/listView/modules/ViewMagazine",
   "ecm/widget/listView/modules/ViewFilmStrip",
   "ecm/widget/listView/modules/DocInfo",
   "ecm/widget/listView/modules/Bar",
   "ecm/widget/listView/modules/Toolbar",
   "ecm/widget/listView/modules/Breadcrumb",
   "ecm/widget/listView/gridModules/DndRowMoveCopy",
   "ecm/widget/listView/gridModules/DndFromDesktopAddDoc",
   "ecm/widget/listView/gridModules/RowContextMenu","dojo/domReady!"],
function( parser, registry, ContentList, ContentItem, ResultSet,
ViewDetail, ViewMagazine, ViewFilmStrip, DocInfo, Bar, Toolbar,
Breadcrumb, DndRowMoveCopy, DndFromDesktopAddDoc, RowContextMenu) {
   parser.parse();
   function getContentListModules() {
      var viewModules = [];
      viewModules.push(ViewDetail);
      viewModules.push(ViewMagazine);
      if (ecm.model.desktop.showViewFilmstrip) {
         viewModules.push(ViewFilmStrip);
      }
      var array = [];
```

```
   array.push(DocInfo);
   array.push({
      moduleClass: Bar,
      top: [
            [
               [
                  { moduleClass: Toolbar},
                  {
                     moduleClasses: viewModules,
                     "className": "BarViewModules"
                  }
               ]
            ],
            [
               [
                  { moduleClass: Breadcrumb}
               ]
            ]
         ]
   });
   return array;
};
function getContentListGridModules() {
   var array = [];
   array.push(DndRowMoveCopy);
   array.push(DndFromDesktopAddDoc);
   array.push(RowContextMenu);
   return array;
};
var contentList = registry.byId("contentList");
contentList.setContentListModules(getContentListModules());
contentList.setGridExtensionModules(getContentListGridModules());

var itemProperties = {
      attributes: { "{NAME}":"DummyFolder"},
      id:"1",
   };
...
```

Although you defined the `itemProperties` structure in the previous example, it is shown here to help you find where to place the code snippet.

See the array of required modules and the corresponding parameter in the callback function of the Dojo required at the beginning. You must declare every module you need in the code.

**Note:** Be careful with the order of the required modules and the corresponding parameter of the callback function: the position of the modules and parameter is crucial and must fit together. For example, the `"dijit/registry"` is at position 2 in the array, and the `registry` parameter is also at position 2 in the callback function.

Invoking the page again has a result similar to Figure 9-15. You will see the icons of the modules on the upper right side. Do not try to switch between the modes immediately, because the dummy item in the result list does not have all the attributes needed for all the views. First, provide a real ResultSet model.

*Figure 9-15 ContentList widget with dummy ResultSet model and different modules*

## 9.7.4 Step 3: Select and initialize the necessary model classes

The main task of this step is to get real data for the ContentList widget. This widget expects real data to be structured as ResultSet model. The content of a folder can be retrieved as such a ResultSet model.

**Note:** To use any of the visual components in an external application, we must prepare the IBM Content Navigator JavaScript model components that are required by the widgets we want to use. The JavaScript Modeling Library is illustrated in Figure 1-16 on page 48.

Repository models represent configured IBM Content Navigator repositories and offer functionality to access and execute actions on the repository objects.

To retrieve the folder and its content for displaying in the ContentList widget, we need the appropriate `ecm.model.Repository` model.

The Desktop model encompasses all other classes of the model and specifies the repositories, features, actions, and viewers that are configured for this desktop instance. That is why we first initialize `ecm.model.Desktop`, which will enable us to get the repository model; see Example 9-19.

*Example 9-19 Getting a real ResultSet in External_ICN.html: Step 3*

```
   require([...,"ecm/widget/dialog/LoginDialog","dojo/domReady!"],
         function(...,LoginDialog) {
   ...
   var contentList = registry.byId("contentList");
   contentList.setContentListModules(getContentListModules());
   contentList.setGridExtensionModules(getContentListGridModules());

   ecm.model.desktop.loadDesktop(null, function() {
      var repository = ecm.model.desktop.getDefaultRepository();
      var retrieveFolder = function () {
         repository.retrieveItem("/", function(folder) {
            folder.retrieveFolderContents(false, function(resultSet){
               contentList.setResultSet(resultSet,folder);
            });
         });
      }
      if (!repository.connected) {
         var initialLoginDialog = LoginDialog.getLoginDialog();
   initialLoginDialog.connectToRepository(repository,retrieveFolder);
      } else {
         retrieveFolder();
      }
   }, false);
});
</script>
```

Consider the following information about the example:

- `loadDesktop(<desktop id>,...)`

  We set <desktop id> to null, which loads the default desktop. If you want to use a different desktop, replace the null with the desktop ID of your choice.

- Anonymous function: This is a callback function of loadDesktop, which is executed when the desktop is loaded:
  - Get the default repository
  - `retrieveFolder`

    This function definition retrieves the folder's content for display. Specifically, it retrieves the folder from the repository and retrieves the

folder content from the callback function. Currently we specify the root folder that we will make configurable later. The retrieveFolderContents function has a callback function that has the folder content in its resultSet parameter. The resultSet parameter is `ecm/model/ResultSet`. Finally, the contentList is initialized with this resultSet.

- `if (!repository.connected)`

  If we are not connected to a repository (`connected` means the repository is loaded and the user is authenticated), we need `LoginDialog`, which we declare at the beginning in the `require` statement.

- `connectToRepository`

  This method tries to perform a direct logon to the repository if it is capable of single sign-on (SSO); otherwise, a login window opens (the show method of the LoginDialog is called implicitly in this case). For the callback function, we provide the retrieveFolder function, which will be called after the repository is connected.

- else { `retrieveFolder()`...

  If connected, we can directly retrieve the folder.

Now you have a working ContentList widget when you invoke your page again. See Figure 9-16.

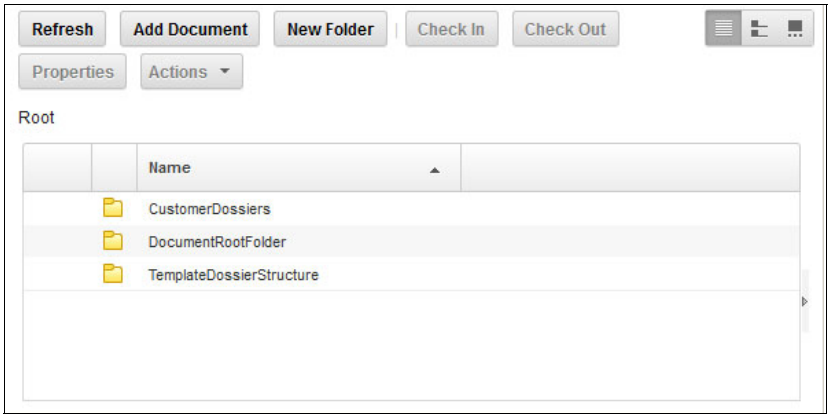



*Figure 9-16 ContentList widget showing the root folder*

After loading the desktop and logging in to the default repository, the root folder is loaded and its content is displayed in the Content List widget. As an option to browse in deeper levels of folders, the Content List widget includes a breadcrumb widget. This widget indicates the actual depth in the folder structure and provides the functionality to navigate back to any folder on the path.

One small item is still missing, and you can see it if you try to open or preview a document, that is you will get an error that the viewer object is null.

**Background:** IBM Content Navigator has implemented a lazy loading pattern for the viewers. The implementation was done in legacy Dojo mode. But for our page, we specified `async=true` in the Dojo configuration object to tell the framework to use the new AMD loader, so to make lazy-load work, we explicitly specify the viewer modules.

We must add the declarations shown in Example 9-20.

*Example 9-20 Adding Viewer module declarations*

```
require([ "dojo/parser","dijit/registry",
   "ecm/widget/listView/ContentList",
   "ecm/model/ContentItem","ecm/model/ResultSet",
   "ecm/widget/listView/modules/ViewDetail",
   "ecm/widget/listView/modules/ViewMagazine",
   "ecm/widget/listView/modules/ViewFilmStrip",
   "ecm/widget/listView/modules/DocInfo",
   "ecm/widget/listView/modules/Bar",
   "ecm/widget/listView/modules/Toolbar",
   "ecm/widget/listView/modules/Breadcrumb",
   "ecm/widget/listView/gridModules/DndRowMoveCopy",
   "ecm/widget/listView/gridModules/DndFromDesktopAddDoc",
   "ecm/widget/listView/gridModules/RowContextMenu",
   "ecm/widget/dialog/LoginDialog",
   "ecm/widget/viewer/FilenetViewer",
   "ecm/widget/viewer/BrowserViewer",
   "ecm/widget/dialog/ContentViewerWindow","dojo/domReady!"],
function(parser, registry, ContentList, ContentItem, ResultSet,
ViewDetail, ViewMagazine, ViewFilmStrip, DocInfo, Bar, Toolbar,
Breadcrumb, DndRowMoveCopy, DndFromDesktopAddDoc, RowContextMenu,
LoginDialog, FilenetViewer, BrowserViewer, ContentViewerWindow) {
```

Now you can view or preview documents with the provided viewers. If you want to use different viewers, add them in an same way.

Finally we make the folder, which will be initially shown in the ContentList widget, configurable through a URL parameter.

Example 9-21 on page 341 shows the definition of the retrieveFolder function after the modifications to read the path of the folder from the **`path`** URL parameter.

*Example 9-21 Implementation of External_ICN.html with configurable folder*

```
require([...,"dojo/io-query","dojo/domReady!"],
   function(...,ioQuery) {

...
var retrieveFolder = function () {
   var path = "/";
   if (window.location.search !== '') {
      var urlparams=
         ioQuery.queryToObject(window.location.search.substring(1)) ;
      if (urlparams.path)
         path = urlparams.path;
   }
   repository.retrieveItem(path, function(folder) {
      folder.retrieveFolderContents(false, function(resultSet){
         contentList.setResultSet(resultSet,folder);
      });
   });
}
```

Consider the following information about the example:

- Add `"dojo/io-query"` and `ioQuery` to the `require` signature.
- The default path is `"/"`, which specifies the root folder.
- The `window.location.search` provides the URL parameters beginning with the question mark.

  For example to get the content of a folder with the `/Customers/L/L_Customer` path, we specify the following information:

  `server:port/navigator/External_ICN.html?path=/Customers/L/L_Customer`

  Where `window.location.search` will be set to `?path=/Customers/L/L_Customer`
- The `ioQuery.queryToObject` transforms the search string to a JavaScript object.
- In `if (urlparams.path)`, we check if the path parameter is set and pass it to the `retrieveItem` function.

Now, if you use the following URL in the browser, you can see the content of the specified folder:

`http://server:port/navigator/External_ICN.html?path=/CustomerDossiers/John%20Smith`

In our case, the path points to a dossier we create in Chapter 4, “Developing a plug-in with basic extension points” on page 117. See Figure 9-17.

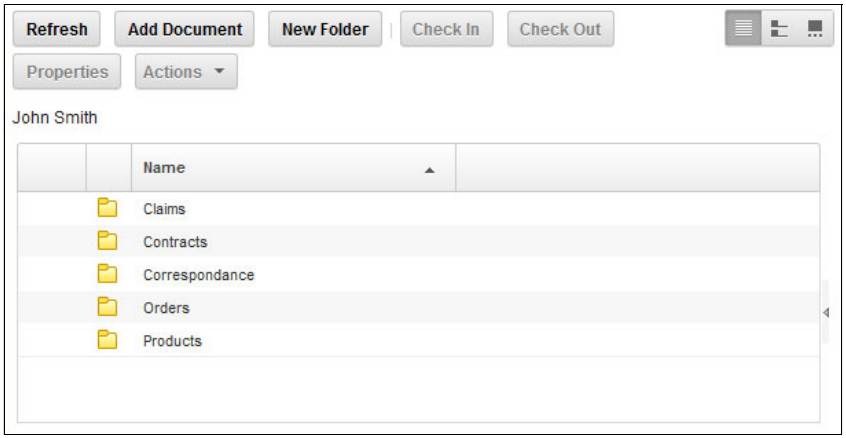



*Figure 9-17 Showing the content of a specific folder through URL parameter*

## 9.7.5 Step 4: Wire the widgets together through event registration

In addition to loading and initializing the model classes, the widgets must be wired together through event registrations. In our sample, most of the wiring is done internally in the ContentList widget itself. So all ContentList modules, for example the breadcrumb widget, are already wired to be able to react to events such as opening a folder, and so on.

One aspect to improve our sample is to add a handler routine if the session expires. Example 9-22 shows the code snippet where we wire `onSessionExpired` event: When a session is expired, IBM Content Navigator calls the onSessionExpired method of the desktop model. With the inner `dojo.connect` we register the sessionExpiredHandler method of the LoginDialog to be invoked, which opens `LoginDialog` to prompt the user to provide credentials again.

*Example 9-22 Connecting a handler for an expired session*

```
if (!repository.connected) {
    var initialLoginDialog = LoginDialog.getLoginDialog();
    initialLoginDialog.connectToRepository(repository,retrieveFolder);
    dojo.connect(initialLoginDialog,"onConnected", function() {
        dojo.connect(ecm.model.desktop, "onSessionExpired",
            initialLoginDialog, "sessionExpiredHandler");
    });
} else {
    retrieveFolder(repository);
}
```

We do not want to register this handler immediately because this might result in two login windows if you invoke this page with a session that is expired. That is why we need the outer dojo.connect (after the first LoginDialog successfully authenticates the user to the repository, it invokes the onConnected methods). With the outer dojo.connect, we register to this event a callback function that invokes the inner dojo.connect.

To test it, invoke your page, and delete the current cookies of your browser. The next attempt to open a folder, which you have not previously opened, will open the LoginDialog; see Figure 9-18.

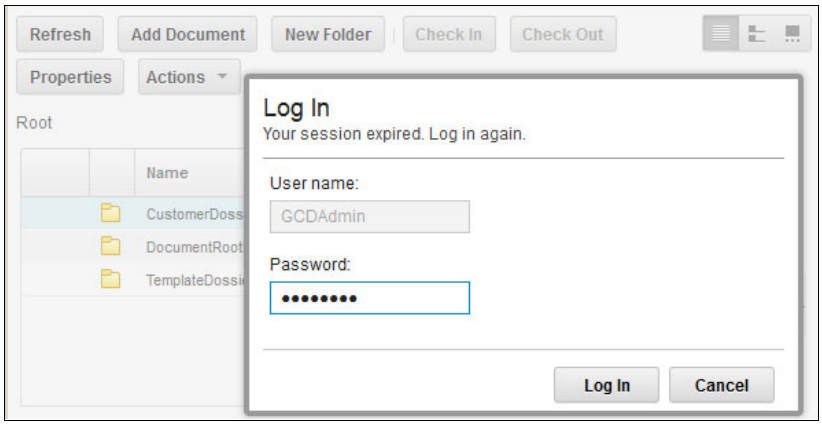



*Figure 9-18 expired Session and the Log In dialog*

Another helpful add-on is displaying error messages. This is especially helpful if a midtier service call fails. For example, when you specify a desktop ID that is not configured in your IBM Content Navigator application, you expect to get a meaningful error message. For this, you must wire an ErrorDialog for displaying error messages. See Example 9-23.

*Example 9-23 Adding an ErrorDialog*

```
require([...,"ecm/widget/dialog/ErrorDialog","dojo/domReady!"],
   function( ...,ErrorDialog) {
      parser.parse();
      var errorDialog = new ErrorDialog();
      dojo.connect(ecm.model.desktop, "onMessageAdded", errorDialog,
"messageAddedHandler");
...
```

You can test this if you temporarily specify a non-existing desktop in this line:

```
ecm.model.desktop.loadDesktop("no-desk-id", function() {..}
```

After reloading the page, you get an error dialog as shown in Figure 9-19.

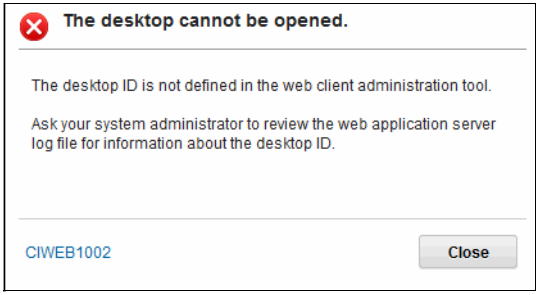



*Figure 9-19 Error dialog*

## 9.8 Integrating the externalized widgets (Step 5)

This is the final step of the integration, which depends on the external application. Before we explore the two main types of integration, the following list summarizes several common aspects:

- Level of integration:
  - Unbound integration:
    - Invoke the prepared URL of IBM Content Navigator directly, which opens a new window where the IBM Content Navigator part is rendered.
    - Invoke the prepared URL of IBM Content Navigator inside an iFrame, which renders IBM Content Navigator as part of the page.
  - Bound integration:

    If you have a container with a runtime environment that differs from a simple web container, for example a Portlet container, and to have a bound integration, you must do more than simply invoke the URL of the externalized widgets of the IBM Content Navigator. To integrate the IBM Content Navigator as a full-fledged part into that environment, you must create a specific artifact of the target environment, for example a Portlet or a web part.
- Authentication:

  In any case, the user who wants to interact with the integrated IBM Content Navigator must be authenticated. So, if you invoke only the prepared URL as-is, a login window opens and the user must provide the credentials.

  Because the user is already authenticated to the external system in normal cases, you do not want to bother this user with another login window. That is

the reason to consider a single sign-on (SSO) solution. IBM Content Navigator is capable of SSO; for the integration, you can follow the same instructions. For more information, go to the following location:

http://pic.dhe.ibm.com/infocenter/cmgmt/v8r4m0/index.jsp?topic=%2Fcom.ibm.installingeuc.doc%2Feucpl012.htm

- Look and feel:

  IBM Content Navigator includes its own Cascading Style Sheets (CSS) that define how IBM Content Navigator is rendered. If you want to adapt the “look and feel” of the integrated IBM Content Navigator, you can, as follows:

  – With the administrative feature of IBM Content Navigator, you can adapt several elements, such as the icons in the icon mapping part, or some general elements, such as the logo, banner, and application name in the Appearance tab of the desktop configuration. See Chapter 2, “Customizing desktop appearance” on page 61.

  – You can change the skins of almost anything by adapting the appropriate Cascading Style Sheets (CSS) or provide your own CSS, which override some of the default settings. However, be aware that the Content Navigator rules are not exposed as API, and might change in future releases.

- Security:

  If the external application and the IBM Content Navigator run on different servers you will face the challenge of the same-origin policy. There are several strategies to cope with this issue, as in the following examples:

  – Configuring a reverse proxy on your HTTP server allows you to access the externalized widgets and the IBM Content Navigator though the same domain (IP) to avoid cross-site scripting issues. This technique is described in 9.8.3, “Defining a reverse proxy in an HTTP Server” on page 347.

  – Set the `document.domain` property to the same value in two windows, so they can interact with each other. This works if the two different servers run on the same subdomain, because then you can set the `document.domain` property to the same base domain, which both servers share.

  – The `window.postMessage` method provides a mechanism to safely enable cross-origin communication. See the following web page:

    https://developer.mozilla.org/en-US/docs/Web/API/window.postMessage?redirectlocale=en-US&redirectslug=DOM%2Fwindow.postMessage

  – If the domains of both servers are in highly trusted zone, for example, both are corporate domains, then Internet Explorer bypasses the same-origin limitation.

## 9.8.1 Deploying the externalized widget (unbound integration)

Figure 9-20 shows the scenario for an unbound integration. Whatever the external application might be (such as another web application, a fat client, or a container like Microsoft SharePoint), the unbound integration will be a URL invocation of the prepared web page (here it is `External_ICN.html`), which externalizes the IBM Content Navigator widgets.

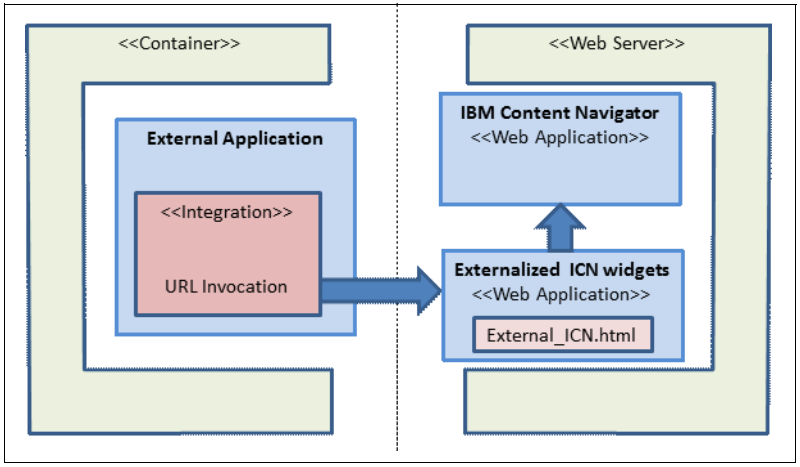



*Figure 9-20 Unbound integration of External_ICN.html*

Our sample page loads JavaScript classes from the deployed and configured IBM Content Navigator instance. To be able to run in the browser without violating the same-origin policy, most modern browsers have the sample that must be deployed on the same application server or HTTP server. Therefore, you may place it in the `war/WebContent` directory of IBM Content Navigator to make it work. Alternatively, you can use another web project to generate your own WAR file and deploy it to the same application server on which your navigator is running.

Exemplary implementations for this type of integration are shown in the following sections:

- 9.9, “Integrating stand-alone widgets in a Microsoft SharePoint page” on page 356
- 9.10, “Integrating stand-alone widgets as a portlet in WebSphere Portal” on page 359

## 9.8.2 Setting up external widgets in the container (bound integration)

Figure 9-21 illustrates the common scenario of a bound integration. The code of the IBM Content Navigator widgets runs directly inside the external application.

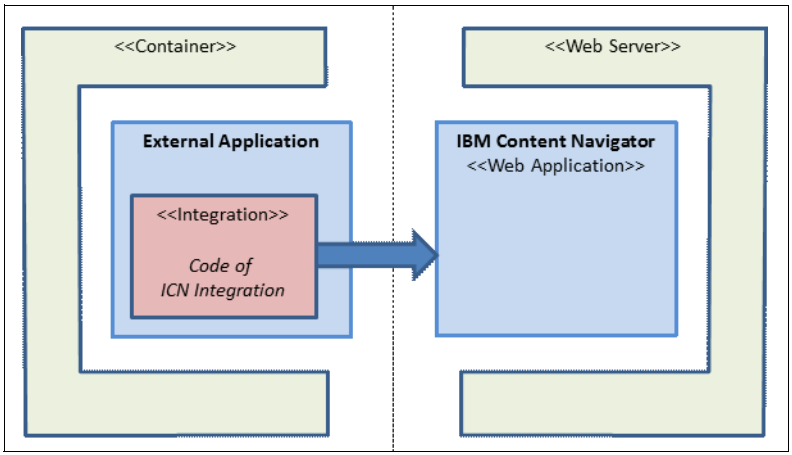



*Figure 9-21 Bound integration of IBM Content Navigator widgets*

The `<<Integration>>` box represents the artifact that must be implemented to run in the target container. In this context, we must set up the IBM Content Navigator widgets in a similar way as we showed in the `External_ICN.html` web page.

As the IBM Content Navigator code runs in the external applications, which is likely to be remote to the web server of IBM Content Navigator, you must cope with security issues such as the same-origin policy. One common solution is described and implemented in the next section.

## 9.8.3 Defining a reverse proxy in an HTTP Server

A new custom web application should run on the same application server as IBM Content Navigator to avoid browser issues with cross-site scripting. But, this is not feasible for a bound integration when the external application runs on a different server than the web server of IBM Content Navigator.

One powerful technique is to treat the externalized Content Navigator widgets as a local deployment by reversing proxy definitions, as illustrated in Figure 9-22 on page 348.

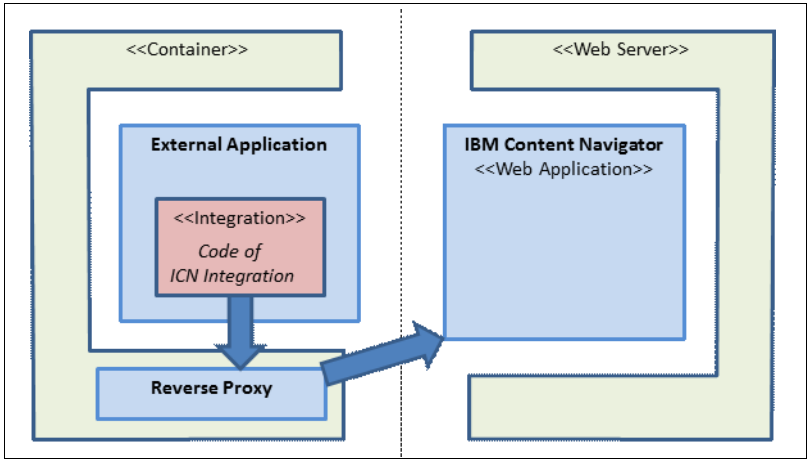



*Figure 9-22 Bound Integration with reverse proxy definition*

On the server of the externalized application, you must install and configure an additional web server, such as the Apache HTTP Server. To let a remote installation of IBM Content Navigator appear as a local one, you must define a reverse proxy in the additional HTTP server.

A reverse proxy is defined in the web server configuration and contains a context and a target URL. If a request reaches the HTTP server with the specified context, the reverse proxy checks to see to which target URL the request should be routed and sends the response to the client. For the client, the response shows up in the same namespace (context) as the request, which is independent from the original location of the called website or application.

In a production environment, the external application usually runs on a dedicated server. A reverse proxy defined in the additional HTTP server configuration routes the requests from the integrated IBM Content Navigator widgets to the remote IBM Content Navigator deployment and simulates that the IBM Content Navigator application and the externalized widgets are running on the same server environment.

The externalized IBM Content Navigator widgets load all resources from the same context as in the following example:

```
<script src="/navigator/dojo/dojo.js.jgz" ....>
```

So the reverse proxy must map the `/navigator` context to the target URL of the deployed IBM Content Navigator application.

To define a reverse proxy, the HTTP server configuration must be changed as follows:

1. Open the `httpd.conf` file in the HTTP server directory under the `<HTTP_Server_InstallPath>/conf/httpd.conf` location.
2. Uncomment the following lines, if they are commented, so they are active:

   ```
   LoadModule proxy_module modules/mod_proxy.so
   LoadModule proxy_http_module modules/mod_proxy_http.so
   ```

3. Add the following lines to the `httpd.conf` file and replace `<server_name>` with the host name of your IBM Content Navigator Server (for example, `contentnavigator.com`) and include the port number in the URL if necessary:

   ```
   ProxyRequests Off
   <Proxy *>
   Order deny,allow
   Allow from all
   </Proxy>
   ProxyPass /navigator http://<server_name>/navigator
   ProxyPassReverse /navigator http://<server_name>/navigator
   ```

4. Restart your HTTP server and try to access the IBM Content Navigator start-page through the following address. Replace `<port>` with the port number of your server (usually 80) and replace `<server_name>` with the host name of the HTTP server.

   ```
   http://<server_name>:<port>/navigator
   ```

   If you can successfully reach IBM Content Navigator then the reverse proxy of the HTTP server has successfully bypassed the local URL (local to the external application) to the remote IBM Content Navigator.

The strict same-origin policy defines the same origin to have the same scheme, host name, and port number. To be compliant with this, the external application should run on the same HTTP server. This is possible if it contains only JavaScript and HTML components, which is unlikely. Otherwise, you must configure the application server of your external application to use this additional HTTP server or define the reverse proxy on the application server itself.

## 9.8.4 Outline of a bound integration

The concrete sample integrations that we show in the next sections are unbound integrations because the bound integrations were not feasible. In this section, however, we provide a bound integration sample so that you have an idea of what the integration looks like. In addition, this sample demonstrates the use of a reverse proxy.

One large disadvantage of unbound integration in previous approaches was that each time you invoked the URL of the externalized IBM Content Navigator widgets, the desktop model was reloaded, which includes a heavy-weight initialization phase.

In this sample, we demonstrate how to initialize the desktop model only once, and after that the user can control the ContentList widget inside the external application: An input field exists where the user can provide a folder path and trigger the retrieval with a button. The folder's content is then displayed in the externalized ContentList widget.

As a starting point, we take the welcome page of the container simulation application in Example 9-2 on page 305. This web page was simulating an unbound integration and will now be adapted to show how a bound integration of an externalized ContentList widget might look.

Example 9-24 shows the code of a web page that emulates an artifact of the external application and has a bound integration of the ContentList widget.

*Example 9-24 Bound integration of ContentList widget in boundIntegration.html*

```
<!DOCTYPE html >
<html>
<head>
   <title>Container for IBM Content Navigator Integration</title>
   <link rel="stylesheet" href="ContainerSimulation.css" media="screen">
   <link rel="stylesheet"href="/navigator/ecm/widget/resources/dojo.css.jgz"
media="screen">
   <link rel="stylesheet" href="/navigator/ecm/widget/resources/ecm.css.jgz"
media="screen">
   <link rel="stylesheet" href="/navigator/ecm/themes/oneui/dojo.css.jgz"
media="screen">
   <link rel="stylesheet" href="/navigator/ecm/themes/oneui/oneui.css.jgz"
media="screen">
   <script>
      dojoConfig = {
         packages:
         [
            {name: "dojo",location: "/navigator/dojo"},
            {name: "ecm",location: "/navigator/ecm"}
         ],
         isDebug: true,
      };
   </script>
   <script src="/navigator/dojo/dojo.js.jgz"></script>
   <script src="/navigator/ecm/ecm.js.jgz"></script>
</head>

<body class="ecm oneui">
<div class="externalAppContent">
```

```
   <input id="folderToShow" type="text" value="/">
   <input id="showFolderButton" type="button" disabled="disabled" value="show
Folder">
</div>
<div id="icnIntegration">
   <div id="contentList" data-dojo-type="ecm.widget.listView.ContentList"
data-dojo-props="plugins:'{dnd:true}',isExternalDND:'true',isResultSetSorted:'true'
,copyOnly:'true'" role="contentinfo" aria-label="Content List" style="width: 100%;
height: 100%;">
   </div>
</div>
<div class="externalAppContent">
   Here is some information from the Container
</div>

<script>
   require([ "dojo/parser", "ecm/widget/listView/ContentList",
      "ecm/widget/listView/modules/ViewDetail",
      "ecm/widget/listView/modules/ViewMagazine",
      "ecm/widget/listView/modules/ViewFilmStrip",
      "ecm/widget/listView/modules/DocInfo",
      "ecm/widget/listView/modules/Bar",
      "ecm/widget/listView/modules/Toolbar",
      "ecm/widget/listView/modules/Breadcrumb",
      "ecm/widget/listView/gridModules/DndRowMoveCopy",
      "ecm/widget/listView/gridModules/DndFromDesktopAddDoc",
      "ecm/widget/listView/gridModules/RowContextMenu",
      "ecm/widget/dialog/LoginDialog",
      "dijit/registry","dojo/io-query",
      "ecm/widget/viewer/FilenetViewer",
      "ecm/widget/viewer/BrowserViewer",
      "ecm/widget/dialog/ContentViewerWindow",
      "ecm/widget/dialog/ErrorDialog","dojo/on",
      "dojo/dom","dojo/dom-attr", "dojo/domReady!"],
   function(parser, ContentList,ViewDetail,ViewMagazine,
ViewFilmStrip,DocInfo,Bar,Toolbar,Breadcrumb,DndRowMoveCopy,
DndFromDesktopAddDoc,RowContextMenu,LoginDialog,registry,ioQuery,
FilenetViewer,BrowserViewer,ContentViewerWindow, ErrorDialog, on,dom,domAttr) {
      parser.parse();
      var errorDialog = new ErrorDialog();
      dojo.connect(ecm.model.desktop, "onMessageAdded", errorDialog,
"messageAddedHandler");
      function getContentListModules() {
         var viewModules = [];
         viewModules.push(ViewDetail);
         viewModules.push(ViewMagazine);
         if (ecm.model.desktop.showViewFilmstrip) {
            viewModules.push(ViewFilmStrip);
         }
         var array = [];
         array.push(DocInfo);
```

```
    array.push({
        moduleClass: Bar,
        top: [
               [
                  [
                     { moduleClass: Toolbar},
                     {
                        moduleClasses: viewModules,
                        "className": "BarViewModules"
                     }
                  ]
               ],
               [
                  [
                     {
                        moduleClass: Breadcrumb
                     }
                  ]
               ]
            ]
    });
    return array;
};
function getContentListGridModules() {
    var array = [];
    array.push(DndRowMoveCopy);
    array.push(DndFromDesktopAddDoc);
    array.push(RowContextMenu);
    return array;
};

var contentList = registry.byId("contentList");
contentList.setContentListModules(getContentListModules());
contentList.setGridExtensionModules(getContentListGridModules());

ecm.model.desktop.loadDesktop(null, function(desktop) {
    var repository = desktop.getDefaultRepository();

    var retrieveFolder = function (path) {
        if (!path) {
            path="/";
        }
        repository.retrieveItem(path, function(rootFolder) {
            console.log("folder retrieved: " + rootFolder);
            rootFolder.retrieveFolderContents(false, function(resultSet){
                contentList.setResultSet(resultSet,rootFolder);
            });
        });
    }

    var initialLoginDialog = LoginDialog.getLoginDialog();
```

```
            var doConnections = function() {
                myButton = dom.byId("showFolderButton");
                on(myButton, "click", function(evt){
                    textInput = dom.byId("folderToShow");
                    if (textInput.value) {
                        retrieveFolder(textInput.value );
                    }
                });
                domAttr.remove(myButton, "disabled");
                dojo.connect(ecm.model.desktop, "onSessionExpired",initialLoginDialog,
"sessionExpiredHandler");
            };

            if (!repository.connected) {
                initialLoginDialog.connectToRepository(repository);
                dojo.connect(initialLoginDialog,"onConnected", function() {
                    doConnections();
                });
            } else {
                alert("Desktop reloaded!");
                doConnections();
            }
        }, false);
    });
</script>

</body>
</html>
```

Major changes to `index.html` are as follows:

- Define a text input field and a button in first `DIV`.
- Define the ContentList widget in the second `DIV`.
- Insert all the JavaScript code to set up the ContentList widget. This is mainly the code we develop in 9.7, “Integrating specific widgets of Content Navigator” on page 324. Modifications to the code of `External_ICN.html` are as follows:
  - Adapt the retrieveFolder function to accept a `path` argument.
  - Add the doConnections function, which does the wiring of events. To set up the wiring of events, use the following steps:
    i. Click **showFolderButton**, which triggers the retrieveFolder method and passes the user input as folder path. The button is enabled after the user is connected to the repository.
    ii. Add a handler for expired sessions.

– Add an alert to see if the page is reloaded. When the page is reloaded, we are already connected to the repository and the else branch is executed.

  This should not happen because we want to control the ContentList widget with a wired button and not by URL invocation and reloading of the whole page.

Adapt the `ContainerSimulation.css` file as shown in Example 9-25.

*Example 9-25 Adapted ContainerSimulation.css for bound integration*

```
.externalAppContent{
   background-color:#ddddff;
   height: 50px;
   text-align: center;
   font-size: 24px;
}
#icnIntegration{
   width: 800px;
   height: 600px;
   background-color:#ddddff;
   margin: 0 auto;
    text-align: center;
}
```

To verify that no errors exist, deploy the Container Simulation application with the new `boundIntegration.html` and the modified CSS file in the application server of IBM Content Navigator. (You can also copy the two files directly into the IBM Content Navigator deployment.) Invoking `boundIntegration.html` in your browser should look like Figure 9-23.

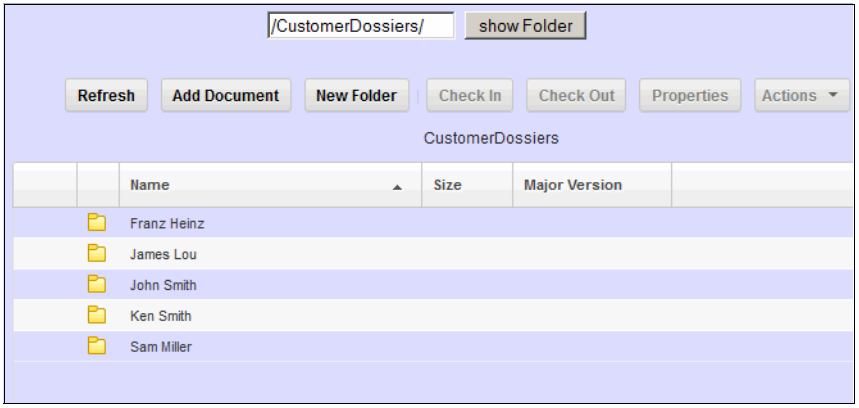



*Figure 9-23 Bound integration of ContentList widget into an external web page*

This was straightforward for several reasons:

- The external application in this case is the Simulation Container application, which is a simple web application: It is not a container with a different runtime environment and different lifecycle of its applications. Both Microsoft SharePoint and IBM WebSphere Portal Server implicate their own runtime environment, in which you must set up the Dojo framework and get the ContentList widget initialized.
- We had total control over the whole web page of integration. In some external containers, you get only a specific place inside a complex web page where you can integrate your code, then it is much more difficult to set up Dojo and your externalized widgets. This is, for example, the case when integrating into Microsoft SharePoint web parts.
- No other Dojo versions are present in the external application so there is no Dojo version conflict. This is the challenge for a bound integration with IBM WebSphere Portal server.

To have a more realistic scenario, we run the code on a remote server and demonstrate how to avoid cross-scripting issues with a reverse proxy.

Follow the instructions (in 9.8.3, “Defining a reverse proxy in an HTTP Server” on page 347) for installing and configuring the additional HTTP server.

Then copy the `boundIntegration.html` file and the adapted version of the `ContainerSimulation.css` file to the `htdocs` directory of the HTTP server; see Figure 9-24.

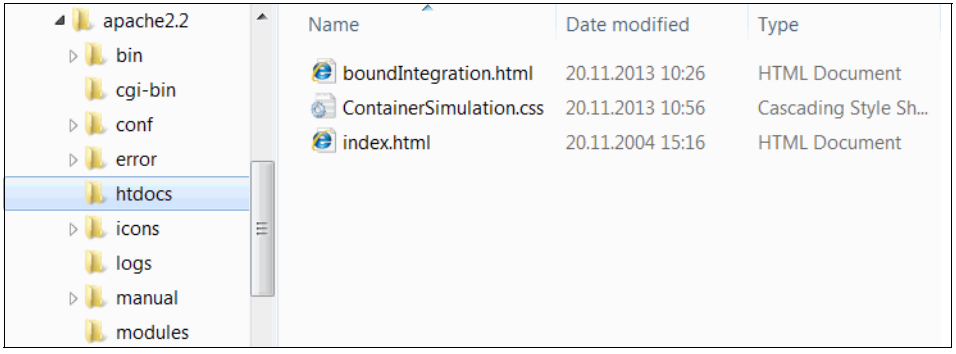



*Figure 9-24 Resulting htdocs directory*

If you start your browser with the following URL, you can see, again, a sample page of the bound integration as shown in Figure 9-23 on page 354:

`http://<server>/boundIntegration.html`

This sample integration will give you a good understanding to realize a bound integration with a concrete external application.

# 9.9 Integrating stand-alone widgets in a Microsoft SharePoint page

The stand-alone widgets can be integrated with software of several third-parties. In this section, we show how to integrate the stand-alone widgets in a Microsoft SharePoint page as an example.

## 9.9.1 Example overview

A fictitious company B uses multiple ECM repositories. The company installed, customized, and extended IBM Content Navigator for direct access and management of their documents. In the previous chapters, we showed you how to customize and extend IBM Content Navigator. Now, the company also wants to enable its Microsoft SharePoint users to access the documents that are currently stored in one of the company's ECM repositories.

So that the Microsoft SharePoint users can access the ECM documents, the company wants to leverage the capabilities of IBM Content Navigator to integrate parts of the IBM Content Navigator user interface into an external system. The approach that is shown in this example is to bring IBM Content Navigator visual widgets, which are provided for browsing the repositories, into a Microsoft SharePoint page.

## 9.9.2 Implementing the Microsoft SharePoint integration

In the previous sections, we describe how IBM Content Navigator can be prepared to show exactly that part that you want to integrate into a container like Microsoft SharePoint:

- Show IBM Content Navigator as-is. Invoke the URL of the IBM Content Navigator start page.
- Invoke a special part of IBM Content Navigator. The API URL provides deep linking capability to show up specific parts of IBM Content Navigator like specific desktop, specific feature, the content of a folder or the content of a document.
- Invoke IBM Content Navigator with a specific feature and with sideChrome URL parameter to show exactly one feature without any bars or banners at the top.

- Invoke IBM Content Navigator with a custom layout that can be used to render exactly the widgets you want to integrate.
- Invoke a web page that initializes the IBM Content Navigator widget you want to integrate.

One main consideration for each IBM Content Navigator integration is whether it can be done as a bound or unbound integration. *Bound* integration in this scenario means to initialize the externalized IBM Content Navigator widgets directly in a web part web page and run the Dojo code inside the web part. *Unbound* integration uses an iFrame in a web part in which the external application runs.

This section describes the latter option and invokes the externalized ContentList widget that is created in 9.7, “Integrating specific widgets of Content Navigator” on page 324.

If you have not completed that section and want to immediately start implementing the Microsoft SharePoint integration, download the web page that is associated with this book. For download instructions, see Appendix D, “Additional material” on page 535. However, before using it, you must deploy it. See 9.8.1, “Deploying the externalized widget (unbound integration)” on page 346 for instructions.

We now assume that you have an HTML page (`External_ICN.html`) that is deployed inside the IBM Content Navigator deployment and that externalizes the ContentList widget of IBM Content Navigator. You can verify this assumption by directly invoking your page in a browser with the following URL:

```
http://<server>:<port>/navigator/External_ICN.html
```

You can integrate the IBM Content Navigator widgets into a Microsoft SharePoint page without writing much, if any, source code that is specific to Microsoft SharePoint. To do the integration, you use an existing web part, which is referred to as Page Viewer, to display the sample web page that contains the Content List widget on a Microsoft SharePoint page. Internally Microsoft SharePoint will embed an iFrame into the web part where the IBM Content Navigator widget will be displayed.

The configuration is simple because you need to edit only the page to which you want to add the IBM Content Navigator browsing function, or create a page with the browsing function:

1. Select the **Page** tab and click **Edit**.

   The window display is converted to the editing mode and new tabs are displayed at the top of the ribbon bar.

2. Select **Insert** → **Web Part** from the Editing Tools section. The new ribbon menu that opens, provides a list of SharePoint web parts that are available to be added to your page.

   The web part can display data from other sources, such as search results or another web page.

3. Select the **Page Viewer** web part from the list of available web parts and click **Add** from the lower right part of the window.

4. It is added to your page. The top right corner of the added web part contains a small arrow, which slides down a menu. Select **Edit Web Part**.

5. From the Page Viewer, select the **Web Page** radio button and click on the three dots. A window opens where you provide the URL. You can also test the URL that you provide. For our example, we enter the following URL:

   `http://<server>:<port>/navigator/External_ICN.html`

6. Expand the Appearance section and enter a title such as `Browse repository`. Set the height to a fixed value of 500 pixels.

7. Click **Apply** to see the result of your configuration. If everything is okay the ContentList widget should be displayed, click **OK**.

8. Click **Save**. See Figure 9-25.

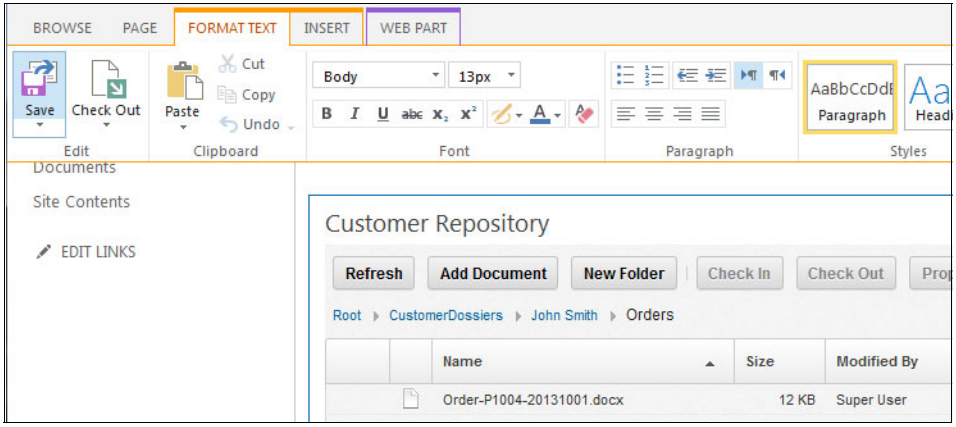



*Figure 9-25 Save the web part*

After the change is made, the page is loaded immediately; it displays the login dialog to the default repository (configured in the accessed default desktop). Log in and the IBM Content Navigator widget is displayed inside the SharePoint page you edited. Figure 9-26 shows the window.

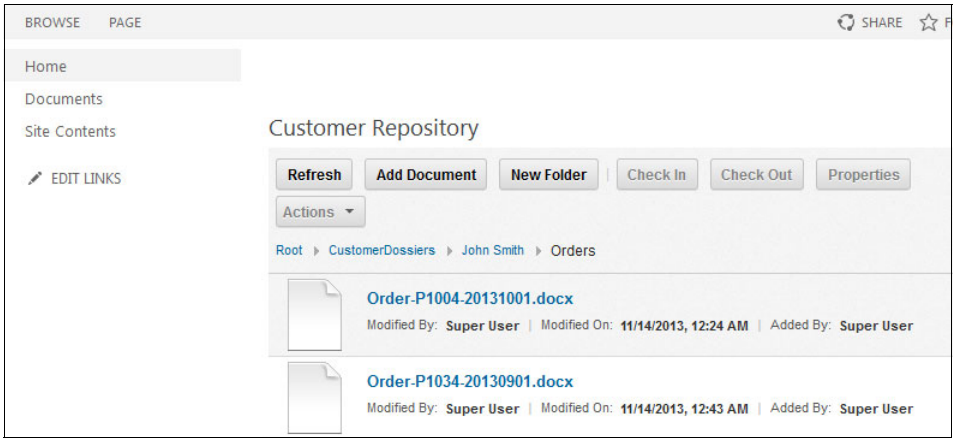



*Figure 9-26 Content List widget on a Microsoft SharePoint page*

The left navigation bar and the top bar belong to Microsoft SharePoint; the main pane displays the integrated IBM Content Navigator widget. This figure shows the integrated widget as a single web part on a page. You may add this web part to an existing page, for example, that shows a document library in a list view, and combine this document library list view with a list view on the repository through the integrated IBM Content Navigator widget.

## 9.10 Integrating stand-alone widgets as a portlet in WebSphere Portal

Integrating IBM Content Navigator into WebSphere Portal requires that you use iFrame. The JavaScript libraries (widgets and models) used by IBM Content Navigator depends on Dojo version 1.8.4, which is not compatible with the Dojo version used by WebSphere portal. Although JavaScript model libraries are using basic Dojo functions and can run also with older portal version of Dojo, it is not supported to use different version of Dojo than the one shipped with IBM Content Navigator. Using iFrame avoid this Dojo compatibility issue and allows you use different versions of Dojo on one page. WebSphere portal is bundled with a Website Displayer portlet that uses iFrame to render specific URL and allows you to seamlessly integrate IBM Content Navigator into WebSphere portal without any coding.

You can use IBM Content Navigator URL API to render desktop, feature, content of the folder, document, or search template in the Website Displayer portlet. In this section, we show how to integrate the HTML page containing the Content List widget created in 9.7, “Integrating specific widgets of Content Navigator” on page 324.

1. Open the page where you want to add the Content List widget and click **Edit Mode**, as shown in Figure 9-27.

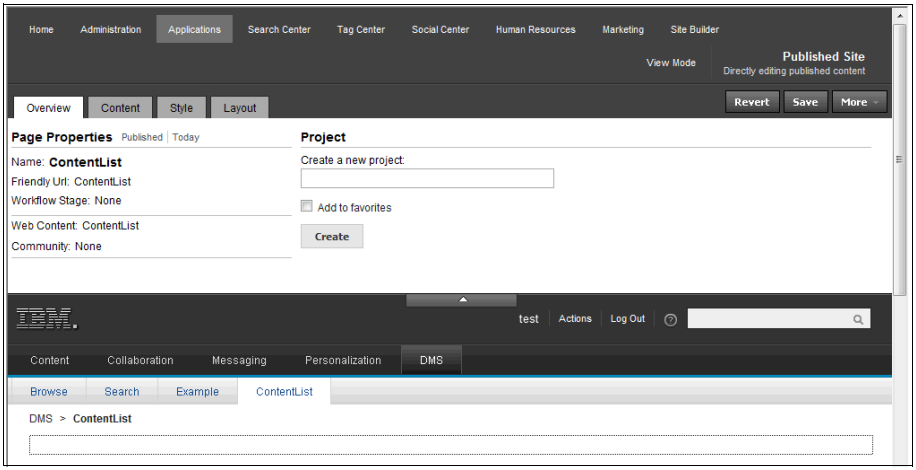



*Figure 9-27 Portal page edit mode*

A new section is rendered on the page showing the tabs with various editing options.

2. Click the **Content** tab and enter `website` into the search field to find the Website Displayer portlet, as shown in Figure 9-28.

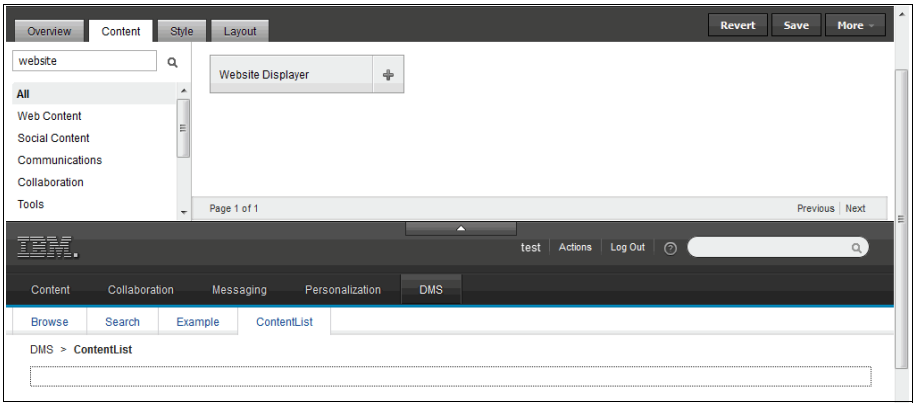



*Figure 9-28 Filtered portlet list*

3. Click the plus sign next to Website Displayer portlet to add the portlet to your page. Then click **Save** to save the page and allow configuration of the portlet. See Figure 9-29.

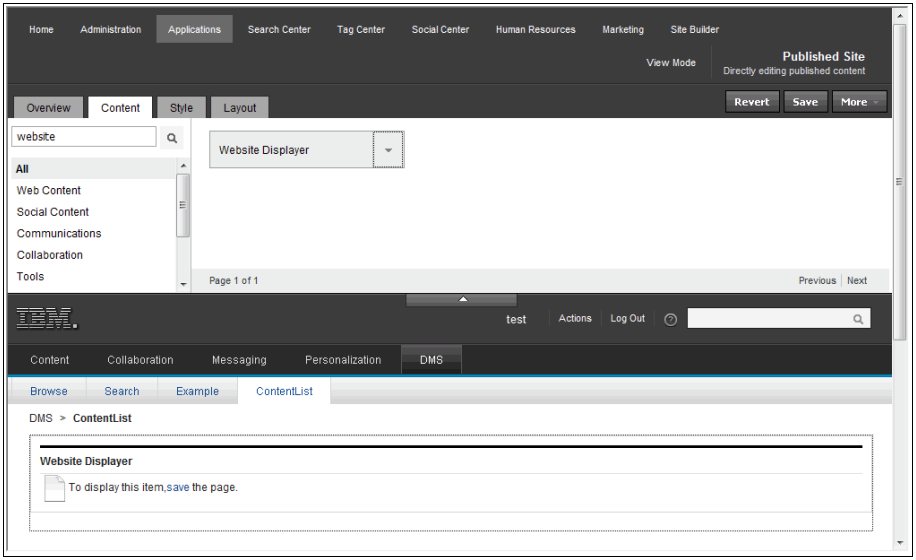



*Figure 9-29 Website Displayer portlet added to page*

4. When the page is saved, you can configure the portlet. Click the small arrow at the top right corner of the portlet and select **Edit Shared Settings**, as shown on Figure 9-30.

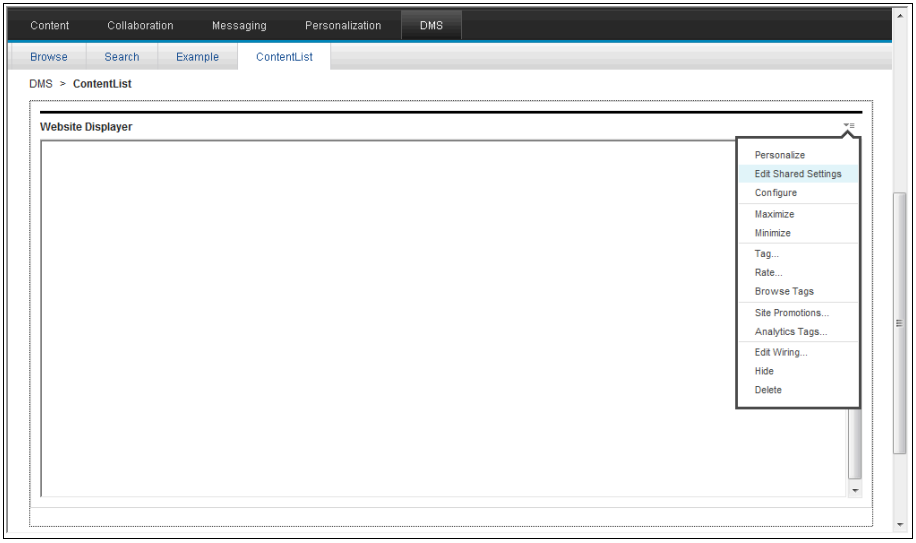



*Figure 9-30 Portlet configuration menu*

5. In the portlet (Figure 9-31), specify the URL to your content list HTML page, and the height of the iFrame where the content renders.

   For example, we use this URL:

   `http://localhost:9080/navigator/External_ICN.html`

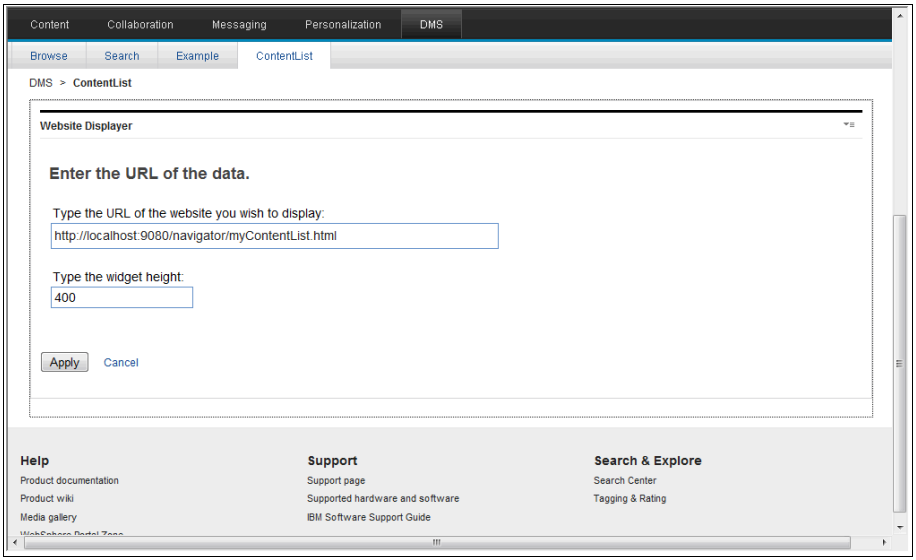



*Figure 9-31 Portlet configuration*

6. Save your settings and return to **View Mode**.

If you configured single sign-on (SSO) between application servers for IBM Content Navigator and for Web sphere Portal, then you see the Content List widget without logging in. Figure 9-32 on page 364 shows the content list widget integrated into portal, where you can use all the functionality of the modules you configured for the ContentList widget, such as breadcrumb navigation, view modules, and context menu.

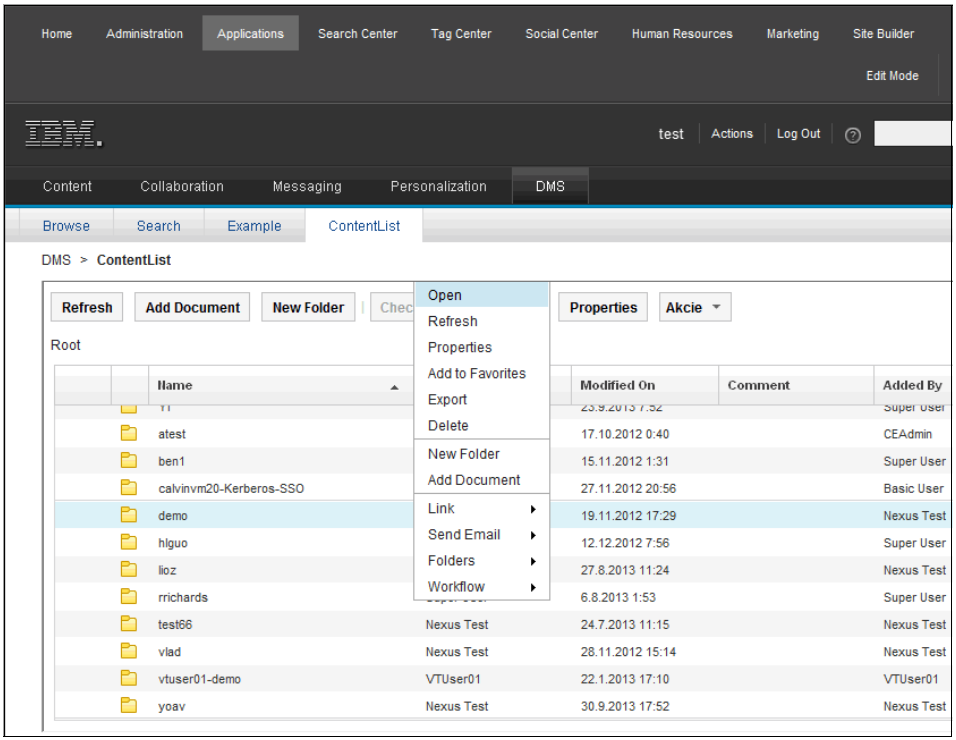



*Figure 9-32 Content list widget integrated into WebSphere Portal*

The default Website Displayer portlet allows you to wire with other widgets on the page. It can receive a whole URL or only a parameter from the URL query string. In this way, you can create a custom portlet that displays a list of documents connected with Website Displayer using the IBM Content Navigator URL API.

## 9.11 Conclusion

This chapter describes how to use IBM Content Navigator widgets externally in other applications. It also shows approaches for externalizing IBM Content Navigator or certain parts of it. This includes developing a custom layout through the layout extension point of IBM Content Navigator and setting up IBM Content Navigator widgets from scratch. The difference of bound and unbound integration is explained and a sample implementation is provided. The integration of IBM Content Navigator widgets into containers is exemplified with Microsoft SharePoint and IBM WebSphere Portal Server.

10

# Customizing built-in viewers and integrating third-party viewers

IBM Content Navigator includes multiple, powerful, and configurable document viewers. The variety of built-in viewers are available to provide seamless viewing support for an expanding list of document types and format that might be stored in your organization's content stores. Integration of commercially available viewers is also possible through the plug-in architecture within IBM Content Navigator. Organizations might want to do this to view specialized file formats, provide additional viewing functions, or use their own preferred viewer.

This chapter provides an overview of the document viewers that are built into IBM Content Navigator. We explore advanced configuration options available in one of the built-in viewers and we review the approach to integrate third-party viewers to IBM Content Navigator.

This chapter covers the following topics:

- Built-in viewers overview
- Example overview
- Customizing the FileNet Viewer
- Exploring viewer extensibility
- Integrating third-party viewers into IBM Content Navigator

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# 10.1 Built-in viewers overview

Within IBM Content Navigator, the ability to view a document without launching that document through its native application is supported by embedding several different viewers into the environment. These viewers provide the ability for users of IBM Content Navigator with the ability to access documents for the sole purpose of reading the content and adding annotations without compromising the source file.

Supporting a variety of content types, specific document viewers are launched when the user has executed the **open** command from the IBM Content Navigator desktop in association with a document. The IBM Content Navigator desktop will launch the appropriate viewing application associated with the document's registered MIME type either because that MIME type has been configured to launch a specific viewer or because the MIME type has not been identified in the configuration, in which case the system will default to a generic, "catch-all" web browser viewer.

A variety of viewers are available to the IBM Content Navigator desktop. These viewers include the AFP to PDF Viewer, AFP Viewer Plugin-Plus Viewer, Applet Viewer, FileNet Viewer, Browser Viewer, and ICC Viewer.

## AFP to PDF Viewer

The AFP to PDF Viewer provides viewer support for the AFP files that may be accessed from an IBM Content Manager OnDemand repository that has been configured for access from the IBM Content Navigator Desktop.

## AFP Viewer Plugin-Plus Viewer

The AFP Viewer Plugin-Plus is a viewer that launches the AFP Viewer plug-in from Content Manager OnDemand Web Enablement Kit. The plug-in must be installed on the client workstation so that this viewer can be available. The `AFPViewerPlugin.jar` file is used to register the plug-in. If the plug-in is not installed on the client workstation, Content Navigator uses the AFP to PDF Viewer instead.

## Applet Viewer

The Applet Viewer provides overlay support for IBM ImagePlus® documents in MODCA format and stored in IBM Content Management libraries.

### FileNet Viewer

The FileNet Viewer is the Daeja ViewONE Pro 4.0.42 as of the release of this publication. With the Daeja Viewer, you can view a vast variety of document formats. It also provides other viewing features such as annotations, redaction, and marking up documents.

### Browser Viewer

The Browser Viewer mapping provides support for many file types and formats and can be used for read-only access to content; other viewer features such as markups or annotations are not required.

### ICC Viewer

The ICC Viewer is intended for items archived through IBM Content Collector such as mail messages, and varying document types that might come through environments like Microsoft SharePoint.

### Overview summary

These viewers are all delivered within IBM Content Navigator through default repository and MIME type mappings and are configured within the Content Navigator desktop through the default viewer map. The system-configured default viewer map is set up to support the most common uses and mappings between viewers and content types. The IBM Content Navigator default viewer map can be accessed through the Administration Desktop. Because it is preconfigured with repository and MIME type mappings, it cannot be edited.

Figure 10-1 shows **Default viewer map** selected in the Viewer Maps section of the Content Navigator configuration.

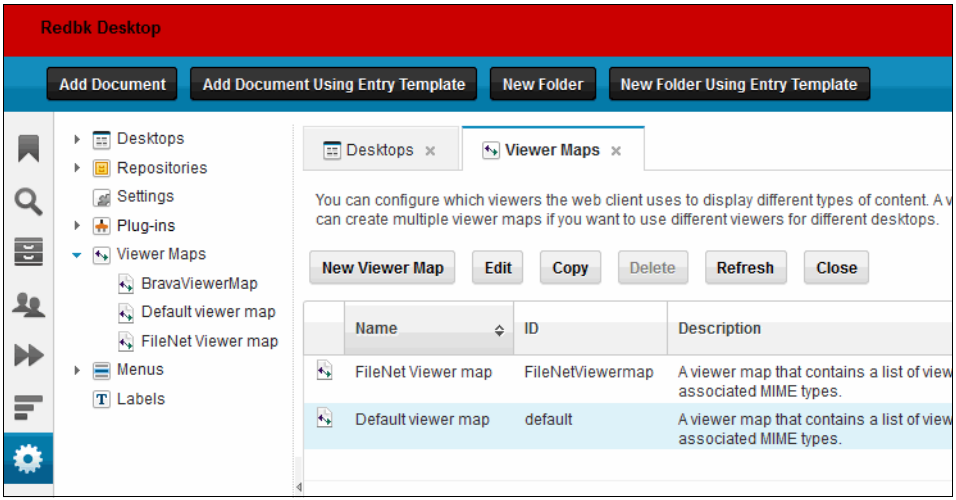



*Figure 10-1 Default viewer map*

Because the default viewer map is a non-editable portion of the Content Navigator configuration, you must copy it if you want to change the default MIME type mappings to reference-specific viewers.

Table 10-1 on page 369 lists the default viewer mappings that are set within the IBM Content Navigator environment.

By changing these mappings, you can override the default viewer mappings within the system if, for example, your organization has an enterprise license to a specific viewer package or platform and you want to use it as the universal viewer for your content management environments.

To determine which mappings you might want to alter, see Table 10-1.

*Table 10-1 Default repository viewer MIME type mappings*

| Repository type | Viewer | MIME types covered |
|---|---|---|
| Content Manager OnDemand | AFP2PDF Conversion<br>Line Data Applet<br>Adobe Reader<br>Applet Viewer | AFP<br>line<br>pdf<br>bmp, gif, jpeg, png, pcx, tif, x-dcx, xpng, vnd.ibm.modcap |
| | Web Browser | All MIME types |
| Content Manager | Adobe Reader<br>Applet Viewer | pdf<br>bmp, gif, jpeg, png, pcx, tif, x-dcx, xpng, vnd.ibm.modcap |
| | Web Browser | All MIME types |
| FileNet Content Manager | FileNet Viewer | pjpeg, jpg, jpeg, bmp, gif, tiff, pdf, png, x-cold, vnd.filenet.image, vnd.filenet.im-cold, vnd.filenet.im-other |
| | Web Browser | All MIME types |
| Content Management Interoperability Services (CMIS) | Applet Viewer | bmp, gif, jpeg, png, pcx, tif, x-dcx, xpng, vnd.ibm.modcap |
| | Web Browser | All MIME types |

For example, you might want to map a common document type (such as a document-formatted (`.doc`) document) to a viewer that allows markups and annotations to be created during viewing. You might want those annotations to be saved with the document into the content management library.

As specified in the MIME type mappings, the web browser viewer is launched as a viewer for document-formatted (`.doc`) documents stored in an IBM FileNet Content Manager system. Because the web browser does not allow for annotations, you might want to map the `.doc` MIME type to a viewer that supports and saves annotations: a viewer such as the FileNet Viewer.

Figure 10-2 on page 370 lists the MIME types (`.doc`, `.xls`, and `.ppt`) added to the MIME type mappings associated with the FileNet Viewer in the custom viewer map.

| Repository Type | Viewer | MIME Type |
| --- | --- | --- |
| FileNet Content Manager | FileNet Viewer | image/pjpeg, image/jpg, image/jpeg, image/bmp, image/gif, image/tiff, application/pdf, application/x-cold, application/vnd.filenet.im-image, application/vnd.filenet.im-cold, application/vnd.filenet.im-other, image/png, application/msword, application/vnd.ms-excel, application/vnd.ms-powerpoint |

*Figure 10-2 Custom MIME type mapping for FileNet Viewer*

Modification of the default MIME type viewer associations requires making a copy of the default viewer map and using the newly created viewer map in your IBM Content Navigator desktop. In cases where a specific MIME type is mapped to more than one viewer in a desktop, the order of the viewers listed in Viewer Maps makes a difference in determining which viewer will be launched. Content Navigator launches the first viewer listed for that particular MIME type.

## 10.2 Example overview

Examples included in this chapter are divided into two categories:

- Those that are done through customizing the FileNet Viewer
- Those that deal with integration of third-party viewers

By customizing the FileNet Viewer, we show these examples:

- Adding headers and footers to documents for branding and confidentiality requirements
- Making annotations anonymous
- Disabling document streaming

In the integration of third-party viewers to IBM Content Navigator, we show the following third-party viewers:

- Snowbound VirtualViewer
- Informative Graphics Brava! Enterprise Viewer

## 10.3 Customizing the FileNet Viewer

In this section, we explore options available in the built-in FileNet Viewer, the Daeja ViewONE Pro Viewer, to enhance the functionality that is available to users of IBM Content Navigator.

### 10.3.1 Adding headers and footers to documents for branding and confidentiality requirements

The Daeja Viewer contains a helpful but overlooked feature that allows you to add a header and a footer to documents rendered by it. Technically, those additions are annotations at the top and bottom of the document pages. The content of the header and footer are customizable and you can set them programmatically. The headers and footers can be placed on documents by using coordinates so that users cannot move or delete them.

One use for adding this header and footer might be to provide disclaimers, from your company, on the documents (viewed through IBM Content Navigator desktop) as part of a legal warning against printing and distribution. To do that in IBM Content Navigator, edit the `filenetViewer_properties.jsp` file, which is located in the applets sub-folder directly under the root of the deployed WAR file. This file bootstraps the Daeja Viewer so you can add the extra configuration there.

To enable headers and footers, use the annotationTemplate directive from the viewer API and also disable FileNet P8 ACL permission support for annotations. Although we can still view and edit annotations on FileNet P8 documents, we cannot change permissions on those annotations. This way applies only for this particular desktop and not for all of IBM Content Navigator.

To add header and footer, complete the following steps:

1. Under the root folder of the WAR file, Create a new file to contain your header and footer content. Name it `headerfooter.txt` file. The text in Example 9-1 on page 293 can be used within your configuration `.txt` file. A sample is also downloadable (see Appendix D, “Additional material” on page 535).

*Example 10-1 Header and footer definition example*

```
[TEXT]
FONTTYPE = arial
FONTHEIGHT = 20
SEMITRANSPARENT = 0
BORDER = 0
TEXT = NOT AN OFFICIAL COPY
X = 0
Y = 0
PAGE = -1
TRANSPARENT = 1
LABEL = Text1
EDIT = 0
```

```
[TEXT]
FONTTYPE = arial
FONTHEIGHT = 20
SEMITRANSPARENT = 0
BORDER = 0
TEXT = This is copyrighted material owned by Company X.  Please
do not distribute.
X = 0
Y = 10.5
PAGE = -1
TRANSPARENT = 1
LABEL = Text1
EDIT = 0
```

2. Add the text shown in Figure 10-2 on page 370 to the `filenetViewer_properties.jsp` file, directly under the check that ensures that the filenetViewerParameters are initialized. An edited sample of the `filenetViewer_properties.jsp` file is available (see Appendix D, “Additional material” on page 535).

*Example 10-2 Enabling headers and footers for documents*

```
annotationTemplate: "/navigator/headerfooter.txt?noCache=" +
Math.random(),
    annotationSubstitution1: "1: <DOCNAME>=" + new Date(),
    annotationSubstitution2: "-1: <DOCNAME>=<EMPTY>",

    customAnnotationToolTip: "Created On: <createdate>",
    alwaysShowCustomAnnotationTooltip: "true",
    filenet: "false",
```

**Note:** We use a random number assigned to the `noCache` variable. The reason is because the `headerfooter.txt` file is a static file and the application server caches it, so, seeing our changes reflected quickly when the file is fetched is difficult. This use of random number is optional. In our content file, we use some variables, such as `<DOCNAME>`. We substitute those with values from the current document being viewed. In this case, we use the `annotationSubstitution` directive to replace the tag with the name of the actual document. We do this for the first page. For the rest, we use the special `<EMPTY>` tag and we keep the field blank. Although that is basically the process, there are many more details available to further customize this feature. For example, instead of using a static file, with the viewer, you can specify a servlet that returns the header and footer, rendering this feature much more powerful.

Figure 10-3 and Figure 10-4 show the results of our example.

*Figure 10-3 The displayed header*

*Figure 10-4 The displayed footer*

## 10.3.2 Making annotations anonymous

The `filenetViewer_properties.jsp` configuration file also defines the hover tips that are displayed for annotations. By default, a user can hover over annotations to see the name of the user who created them. This behavior is expected but not always desirable. Sometimes, a requirement might be for users *not* to know who created the annotation. For example, suppose a loan applicant sees a note on a loan application that clarification is needed for a particular entry. If the applicant also sees, on that note, the name and email address of the user who created the note annotation, the applicant might send email to that person directly rather than to customer service email address. This bypassing of the appropriate communication channels causes a degradation of the process. For security reasons, we do not want that to happen. Therefore, we want every user to log in with a proxy account so that all annotations are displayed as created by that anonymous proxy user.

Figure 10-5 shows a tooltip with the name of the annotator (P8Admin), and the annotation creation date and time, when you hover over an annotation.

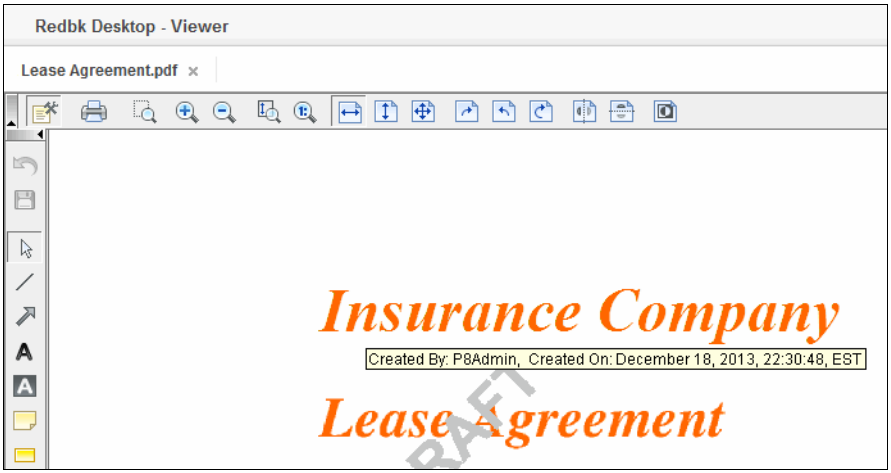



*Figure 10-5 Document showing the name of the annotator'*

To make the annotator name anonymous, the built-in Daeja Viewer provides a way to change the format of the tooltip programmatically. This means we can narrow the scope of this function to a particular desktop or maybe a specific object store, or even only a specific document class.

To make the name of annotator be anonymous, complete the following steps:

1. Locate the `filenetViewer_Properties.jsp` file, which is in the applets directory of the WAR path for the deployed IBM Content Navigator.
2. To the structure, add the tags you want, without the name of the annotator. The Daeja Viewer documentation has information about special tags to use to inject context-aware information to the tooltip.

   For our example, we used only the creation date as the annotation tooltip. Example 10-3 shows the changes for the filenetViewerParameters structure.

   *Example 10-3 Changing annotations tooltip*

```
filenetViewerParameters.customAnnotationToolTip = 'Created on:
<createdate>';
filenetViewerParameters.alwaysShowCustomAnnotationToolTip = 'true';
```

Now, when a user hovers the mouse over the annotation, the name of the annotator is not displayed in the tooltip. See Figure 10-6.

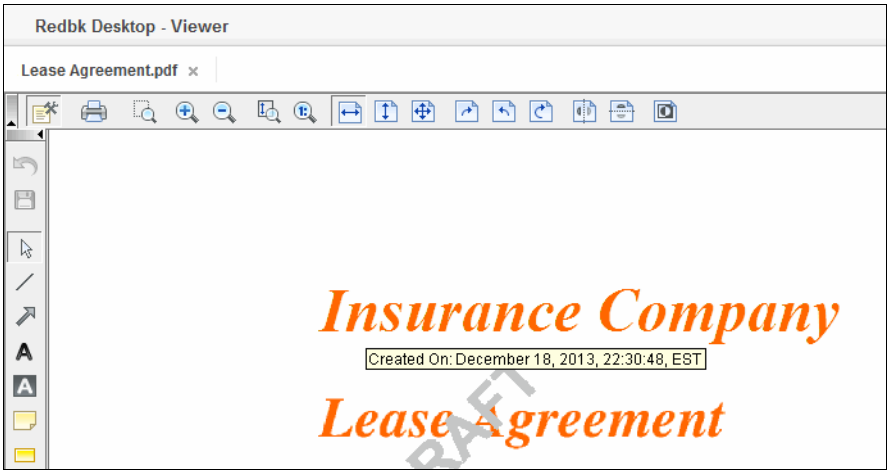



*Figure 10-6 Annotation without annotator's name*

## 10.3.3 Disabling document streaming

Document streaming is enabled in the system for TIF and PDF files, by default. The ability to stream documents can be a valuable feature that allows the user to access content from large files as quickly as possible and to begin viewing or reading the information before the entire document is downloaded.

In some cases, where a proxy server and a load balancer are deployed within the IBM Content Navigator environment, organizations might want to disable document streaming to alleviate issues that might occur within the proxy and load balancer architecture and that can interfere with the downloading of a document.

In IBM Content Navigator, you can disable document streaming if required:

1. Find the `filenetViewer.js file` in the js directory of the WAR path for the deployed IBM Content Navigator.
2. Edit the `navigator/js/filenetViewer.js` file.
3. Find the following text in the `filenetViewer.js` file near line 50 in the file:

   ```
   this.useStreamer = (contentType.indexOf("tiff") != -1 ||
   contentType.indexOf("pdf") != -1);
   ```

4. Modify the line with `this.useStreamer = false` as follows:

   ```
   this.useStreamer = false; // (contentType.indexOf("tiff") != -1 ||
   contentType.indexOf("pdf") != -1);
   ```

After saving the file, document streaming will be disabled for TIF and PDF files in the IBM Content Navigator environment.

## 10.4 Exploring viewer extensibility

This section explores what IBM Content Navigator exposes to developers to integrate viewing solutions.

### 10.4.1 Viewer architecture

Viewing solutions are being built in many ways, but two models are prevalent: viewers that talk to repositories directly from the client-side, and others that rely on a server-side component. Figure 10-7 reflects both architectural styles. Viewer1 exhibits only client-side architecture; Viewer2 relies on a server-side component.

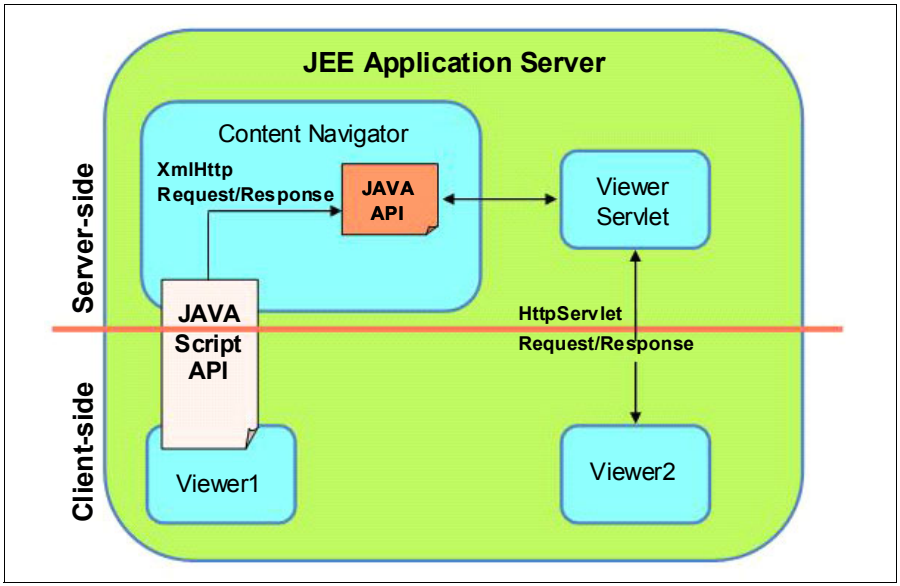



*Figure 10-7 Client-side viewer (Viewer1) and server-side viewer (Viewer2) architecture*

IBM Content Navigator supports both models. If your viewer relies on a server-side component, you can pass information to a servlet and then use repository APIs (for IBM FileNet Content Manager or IBM Content Manager OnDemand, for example) to fetch content and pass it back to your client. Similarly, if your viewer operates completely on the client-side, you can use the

IBM Content Navigator JavaScript modeling layer to fetch content from repositories through the navigator and back down to your client.

With both models, the effort to integrate the viewer into IBM Content Navigator is similar. For client-side viewers, developers must consult the API documentation that is provided by IBM to understand how to use the JavaScript modeling layer to fetch content. This section focuses on how to develop the plug-in for any viewer, independent of its architecture.

## 10.4.2 Extension points

Three classes must be extended to register a new viewer in IBM Content Navigator:

- com.ibm.ecm.extension.Plugin
- com.ibm.ecm.extension.PluginService
- com.ibm.ecm.extension.PluginViewerDef

In those classes, you communicate to the IBM Content Navigator, the behavioral characteristics, and capabilities of your viewer, and also specify how to launch it.

**Note:** to Set up your development environment and project to allow for plug-in development, see Chapter 3, “Setting up the development environment” on page 73.

Other files you need or might want to add to the system to make it work are as follows:

- `ConfigurationPane.html`
- `ConfigurationPane.js`
- `MANIFEST.MF`

### Required components

The classes in this section must be extended to register a new viewer.

#### *The com.ibm.ecm.extension.Plugin class*

By extending this class, you can specify the name of the plug-in (and therefore the name of the viewer), its version, and the Dojo class that corresponds to its configuration user interface. This is explained in 10.5, “Integrating third-party viewers into IBM Content Navigator” on page 387.

Example 10-4 shows the class signature that extends the plug-in.

*Example 10-4 Class signature that extends Plugin*

```
public class MyViewerPlugin extends com.ibm.ecm.extension.Plugin
```

To declare that the plug-in contains viewer functionality, and that there is also a server-side component to the plug-in, add the code shown in Example 10-5.

*Example 10-5 Declare viewer capabilities in plug-in*

```
public PluginViewerDef[] getViewers() {
   return new PluginViewerDef[] { new MyViewerPluginViewerDef() };
}

public PluginService[] getServices() {
   return new PluginService[] { new MyViewerPluginService() };
}
```

To specify the name of the widget that is paired with this plug-in, you must specify the methods shown in Example 10-6.

*Example 10-6 Bind the plug-in to a widget UI configuration component*

```
public String getDojoModule() {
   return "myViewerPluginDojo";
}

public String getConfigurationDijitClass() {
   return "myViewerPluginDojo.ConfigurationPane";
}
```

### *The com.ibm.ecm.extension.PluginService class*

This class must be extended to bootstrap and eventually execute the viewer. This class also handles saving the configuration. Example 10-7 shows the class signature that extends the PluginService class.

*Example 10-7 Class signature that extends PluginService*

```
public class MyViewerPluginService extends
com.ibm.ecm.extension.PluginService
```

The main method, execute, must be implemented. In this method, you can divide the responsibilities of the classes into retrieving and saving the configuration, and opening the viewer, as Example 10-8 on page 379 demonstrates.

*Example 10-8 Extending the execute() method for basic usage*

```
public void execute(PluginServiceCallbacks callbacks,
      HttpServletRequest request, HttpServletResponse response)
      throws Exception {
   String api = request.getParameter("api");
   if (api.equals("getConfiguration"))
      executeGetConfiguration(callbacks, request, response);
   else if (api.equals("setConfiguration"))
      executeSetConfiguration(callbacks, request, response);
   else if (api.equals("openViewer"))
      executeOpenViewer(callbacks, request, response);
}
```

In “Developing the IBM Content Navigator Plug-in for this viewer” on page 390, we demonstrate a fully functioning example of this class. In that section, we describe what type of information is made available in the execute() method to help you bootstrap your viewer.

First, you see from the formal parameters of the method that the entire `HttpServletRequest` class is available. This means that you can get information, such as session ID and server name. Specific to your viewer is information in the request parameters, which you can get through the `request.getParameterMap()` method. In that map, you find information such as the document ID that is to be opened and also the name of the repository in which the document resides. The following information is available in the map:

- docId: The unique identifier of the document.
- docName: The name of the document.
- repositoryId: The name of the repository in which this document resides. This name is the name as it is configured in the IBM Content Navigator, not the name of the repository in the underlying archive.
- mimeType: The MIME type of the document.
- serverType: The type of the underlying archive (p8, cm, od).
- vsId: The ID of the version series of the document.
- replicationGroup: The name of the replicationGroup setup through IBM Content Federation Services (CFS), for bidirectional support between IBM FileNet Content Manager and IBM FileNet Image Services.
- objectStoreName: The name of the underlying archive in which the document is located.

- docUrl: A string that contains the exact parameters that are needed to get the document that uses IBM Content Navigator's getDocument functionality. It contains some of the same information (such as vsId, for example) plus new information. See Example 10-9.

*Example 10-9 Information contained in docUrl*

```
/navigator/p8/getDocument.do?docid=Contract%2C%7B62275E2C-AC20-4071-
A57D-9948FB3EE3C3%7D%2C%7B827AC031-8AF3-4FCC-82FC-3F2C9C551F5B%7D&te
mplate_name=Contract&repositoryId=MyCompanyRepository&vsId=%7B08F39A
DF-7637-4F07-93D3-8AE0C908FC86%7D&objectStoreName=MyObjectStore&secu
rity_token=-5225665686370023147
```

- privs: The privileges that the viewer must honor. The following privileges are passed down to the viewer, based on the disposition of the document:
  - printDoc: Ability to print the document. Invoking print from the browser might not be preventable but it also does not result in a fully printed document.
  - exportDoc: Ability to save the document to local disk storage.
  - editDoc: Ability to edit the document, for example rearrange the pages of a PDF.
  - viewAnnotations: Ability to view the annotations on the document.
  - editAnnotations: Ability to add, edit, or remove annotations on the document.

Example 10-10 lists all of these parameters from an actual document.

*Example 10-10 A full example of the parameters passed into the viewer plug-in*

```
key: editDoc value: false
key: objectStoreName value: MyObjectStore
key: replicationGroup value: undefined
key: plugin value: MyViewerPlugin
key: mimeType value: application/pdf
key: editAnnotations value: false
key: serverType value: p8
key: api value: openViewer
key: viewAnnotations value: false
key: action value: MyViewerService
key: docId value:
Contract,{62275E2C-AC20-4071-A57D-9948FB3EE3C3},{827AC031-8AF3-4FCC-82FC-3F2C9C551F
5B}
key: security_token value: -5225665686370023147
key: repositoryId value: MyCompanyRepository
key: exportDoc value: true
key: contentType value: application/pdf
key: printDoc value: true
```

```
key: vsId value: {08F39ADF-7637-4F07-93D3-8AE0C908FC86}
key: docUrl value:
/navigator/p8/getDocument.do?docid=Contract%2C%7B62275E2C-AC20-4071-A57D-9948FB3EE3
C3%7D%2C%7B827AC031-8AF3-4FCC-82FC-3F2C9C551F5B%7D&template_name=Contract&repositor
yId=MyCompanyRepository&vsId=%7B08F39ADF-7637-4F07-93D3-8AE0C908FC86%7D&objectStore
Name=MyObjectStore&security_token=-5225665686370023147
key: docName value: Agreement.pdf
```

As you can see, docUrl is mostly a composition of pieces that are already offered in other parameters. The docId parameter consists of a 3-tuple: the symbolic name of the document class of the document, the identifier of the object store, and the identifier of the document itself.

**Tip:** You can use the following method to help you debug code in your plug-in:

`com.ibm.ecm.serviceability.Logger.logDebug(...)`

The debug statements are in the `system.out` log file of the application server.

### *The com.ibm.ecm.extension.PluginViewerDef class*

This class provides two functions:

- Specifies which repository types the viewer supports (IBM Content Navigator enforces it) and for which document MIME types it is optimized.
- Defines how to launch the viewer by specifying the launch URL pattern.

Example 10-11 shows the class signature that extends the PluginViewerDef class.

*Example 10-11 Class signature that extends PluginViewerDef*

```
public class MyViewerPluginViewerDef extends com.ibm.ecm.extension.PluginViewerDef
```

To specify the repository and MIME types that are supported by your viewer, you can use getSupportedContentTypes() and getSupportedServerTypes() methods, as shown in Example 10-12.

*Example 10-12 Code snippet that specify repository and MIME type support by viewer*

```
public String[] getSupportedContentTypes() {
   return new String[] { “image/tiff”, “image/png” };
}

public String[] getSupportedServerTypes() {
   return new String[] { “p8”, “cm” };
}
```

## Optional components

Optionally, you can also specify a widget that renders the user interface and that collects all configuration information for the viewer. For example, if you want to specify whether the viewer should render annotations on a system-wide scope, you create two files:

- ConfigurationPane.html
- ConfigurationPane.js

We also create a manifest file.

### *The ConfigurationPane.html file*

In this HTML file, you specify the presentation layer components of your configuration user interface. See Example 10-13.

*Example 10-13 Configuration user interface specified in HTML*

```
<div>
   <table class="propertyTable" role="presentation">
      <tr>
         <td class="propertyRowLabel"><span class="required">*</span>
<label
            for="annotationsEnabled">Annotations
Enabled:</label> </td>
         <td class="propertyRowValue">
            <div id="annotationsEnabled"
dojoAttachPoint="_annotationsEnabled"
               dojoAttachEvent="onKeyUp: _onParamChange" maxLength="5"
               dojoType="ecm.widget.ValidationTextBox" required="true"
trim="true"
               propercase="false"></div>
         </td>
      </tr>
   </table>
</div>
```

### *The ConfigurationPane.js file*

In this JavaScript file, you specify the validation logic for your configuration parameters. See Example 10-14.

*Example 10-14 Validation logic defined in JavaScript*

```
dojo.provide("myViewerPluginDojo.ConfigurationPane");

dojo.require("ecm.widget.admin.PluginConfigurationPane");
dojo.require("ecm.LoggerMixin");
```

```
dojo.require("dijit._Templated");

/**
 * @name myViewerPluginDojo.ConfigurationPane
 * @class
 * @augments ecm.widget.admin.PluginConfigurationPane
 */
dojo
      .declare(
            "myViewerPluginDojo.ConfigurationPane",
            [ ecm.widget.admin.PluginConfigurationPane, dijit._Templated,
                  ecm.LoggerMixin ],
            {
               templateString : dojo.cache("myViewerPluginDojo",
                     "templates/ConfigurationPane.html"),
               widgetsInTemplate : true,
               configuration : null,

               constructor : function() {
               },

               postCreate : function() {
                  this.inherited(arguments);

                  this._annotationsEnabled.invalidMesssage = "Please set the
value to true or false";
s
                  this._annotationsEnabled.isValid = dojo
                        .hitch(
                              this,
                              function(isFocused) {
                                 return this

._validateAnnotationsEnabled(isFocused);
                              });
               },

               loadConfigurationString : function(configurationString) {
                  this.configuration = eval("(" + configurationString
                        + ")");
                  if (!this.configuration.annotationsEnabled) {
                     this.configuration.annotationsEnabled = "";
                  }
               },

               load : function(callback) {
                  var methodName = "load";
                  this.logEntry(methodName);
```

```
                    if (this.configurationString) {
                       this
.loadConfigurationString(this.configurationString);

                       this._annotationsEnabled.set('value',
                             this.configuration.annotationsEnabled);
                    } else {
                       // defaults
                       this._annotationsEnabled.set('value', 'false');
                    }

                    this.logExit(methodName);
                 },

                 _onParamChange : function() {
                    var methodName = "_onParamChange";
                    this.logEntry(methodName);

                    if (this.configuration == null)
                       this.configuration = new Object();

                    this.configuration.annotationsEnabled =
this._annotationsEnabled.get('value');

                    this._annotationsEnabled.validate();
                    this.configurationString =
JSON.stringify(this.configuration);
                    this.onSaveNeeded(true);
                    this.logExit(methodName);
                  },

                 _validateAnnotationsEnabled : function(isFocused) {
                    var methodName = "_validateAnnotationsEnabled";
                    this.logEntry(methodName);

                    var valid = true;
                    var annotationsEnabled = this._annotationsEnabled
                          .get('value');

                    if (annotationsEnabled.length <= 0) {
                       valid = false;
                    } else {
                       if (annotationsEnabled == 'true'
                             || annotationsEnabled == 'false') {
                          valid = true;
                       } else {
                          valid = false;
                       }
```

```
            }

            this.logExit(methodName);

            return (valid);
        },

        validate : function() {
            var methodName = "validate";
            this.logEntry(methodName);

            var valid = this._annotationsEnabled.isValid();

            this.logExit(methodName);

            return valid;
        }
    });
```

Figure 10-8 shows the rendered configuration, based on our example.

*Figure 10-8 The configuration UI rendered*

It is important to reflect on the concepts here. Other products simply allow you to specify a property name/value pair for configuration, and keep you hoping the user will not enter something that is not allowed. With IBM Content Navigator, however, you can both specify exactly how you want the configuration user interface to look and program the logic that governs that user interface. This feature is powerful.

IBM Content Navigator provides a validation widget for a simple text box, named `ecm.widget.ValidaionTextBox` (as used in Example 9-16 on page 306). To compose a more complex user interface, such as a combo box or a slider, you must extend the form widget that is available in the Dojo framework. For example, we outline an example for a combo box. We create a new widget that has the name `mycompany.widget.ValidationComboBox`. We present the outline and use the existing `ecm.widget.ValidationTextBox` as reference.

Example 10-15 shows the code snippet for the JavaScript.

*Example 10-15 ValidationComboBox.js*

```
dojo.provide("mycompany.widget.ValidationComboBox");

dojo.require("dijit.form.ComboBox");
dojo.require("ecm.widget._HoverHelpMixin");
dojo.require("mycompany.Messages");

dojo.declare("mycompany.widget.ValidationComboBox", [
dijit.form.ComboBox, ecm.widget._HoverHelpMixin ], {
/* fill in implentation details here */
});
```

You do not have to specify an accompanying HTML file. Your widget uses the HTML file of the original widget to render itself. After you complete the implementation, you can use the newly created widget in your `ConfigurationPane.html` file as shown in Example 10-16.

*Example 10-16 Snippet for ConfigurationPane.html that includes a combo box*

```
<div id="annotationsEnabled" dojoAttachPoint="_annotationsEnabled"
   dojoAttachEvent="onChange: _onParamChange"
   dojoType="mycompany.widget.ValidationComboBox"
   required="true"
</div>
```

Make sure to examine the events that are available by the widget you are extending so that you can wire the control correctly to your validation layer. For instance, in Example 10-16, we use the onChange event to be notified when the user selects a value from the combo box.

### *The MANIFEST.MF file*

We create a manifest file to use when we create our JAR file. The minimum configuration tags are shown in Example 10-17.

*Example 10-17 MANIFEST.MF*

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 2.4 (MyCompany)
Plugin-Class: com.mycompany.viewerplugin.MyViewerPlugin

Name: build
Built-By: developer
Build: ${TODAY}
```

**Tip:** Although not shown in this example, we suggest that for production environment plug-ins, you prepend all ID attributes of HTML tags with a unique identifier. This way, you ensure that the plug-ins do not contain tags with the same ID as other plug-ins, which can cause many issues.

## 10.5 Integrating third-party viewers into IBM Content Navigator

In this section, we explore integrating two separate third-party viewers with IBM Content Navigator. Both viewers provide enhanced capabilities to the list of built-in viewers and can enrich the tools available to users of IBM Content Navigator.

Typically, when implementing a third-party viewer that is not built in to IBM Content Navigator, you are required to install or deploy the third-party viewer based on the manufacturer's instructions. The approach to integrating a third-party viewer into the Navigator environment requires a plug-in that can be used within the IBM Content Navigator architecture to call the third-party viewer. The viewer plug-in, or connector, is responsible for calling the external viewer.

### 10.5.1 Snowbound VirtualViewer

The first viewer we review is the Snowbound VirtualViewer, which is based on Ajax. The viewer does not require client-side installation or assumptions about what software (for example JRE version) is running on the user machine, other than a web browser. The viewer is built to rely on a server-side servlet, which

delivers document content to optimize viewing on the client-side. This viewer includes features such as page-reordering, annotations-manipulation, and redactions.

This section has an overview of the prerequisites required to set up the Snowbound VirtualViewer environment and the details required to set up the VirtualViewer Content Navigator Connector. The Content Navigator Connector provided by Snowbound is a commercially available plug-in, developed by Snowbound, the manufacturer of the virtual viewer. This section also describes the details behind developing a connector using Snowbound as an example.

## Prerequisites

Before we start, certain preparation work must be completed. The Snowbound VirtualViewer contains instructions for how to install the product. We have the following assumptions:

- The Snowbound VirtualViewer is installed in a subdirectory directly under the navigator WAR folder. In our setup, this folder is as follows:

  `C:\<WAS_INSTALL>\AppServer\profiles\AppSrv01\installedApps\P8Node01Cell\navigator.ear\navigator.war`

  Under this folder, we add the folder that contains the Snowbound VirtualViewer files. We named this folder `VirtualViewer`.

- Based on the Snowbound installation instructions, the necessary entries are added to the `web.xml` file. In our setup, it is in the following location:

  `C:\<WAS_INSTALL>\AppServer\profiles\AppSrv01\config\cells\P8Node01Cell\applications\navigator.ear\deployments\navigator\navigator.war\WEB-INF`

Pay close attention to the parameters shown in Example 10-18. Be sure that you replace the bold text with the host name or IP of your server.

*Example 10-18 Snowbound's servlet configuration*

```
<servlet>
   <servlet-name>AjaxServlet</servlet-name>
   <servlet-class>com.snowbound.ajax.servlet.AjaxServlet
   </servlet-class>
   <init-param>
      <param-name>contentHandlerClass</param-name>

<param-value>com.snowbound.snapserv.servlet.NavigatorContentHandler
      </param-value>
   </init-param>
   <init-param>
      <param-name>codebase</param-name>
      <param-value>http://<ip/hostname>/navigator/VirtualViewer
      </param-value>
   </init-param>
   <init-param>
      <param-name>servletURL</param-name>
      <param-value>http://<ip/hostname>/navigator/VirtualViewer
      </param-value>
   </init-param>
</servlet>

<servlet-mapping>
   <servlet-name>AjaxServlet</servlet-name>
   <url-pattern>/VirtualViewer/AjaxServlet</url-pattern>
</servlet-mapping>
```

- The `VirtualViewer/config.js` file is modified to reflect the correct servlet path. Example 10-19 shows our setup.

*Example 10-19 config.js file modifications*

```
var servletPath =  "/navigator/VirtualViewer/AjaxServlet";
```

**Tip:** If you have problems with the installation, a beneficial step might be to try to install the viewer for WorkplaceXT by following the exact instructions that are provided. This step can help you become accustomed to the installation process.

## Developing the IBM Content Navigator Plug-in for this viewer

Developing the plug-in for this viewer is not much different from previous examples. In this section, we use the same JavaScript files because not much plug-in-level configuration must be done.

We create a Java project in Eclipse (or Rational Application Developer) and add the following JAR files to the build path:

- navigator.jar
- JSON4J.jar
- j2ee.jar

Next, we add the following source code files:

- MyViewerPlugin
- MyViewerPluginService
- MyViewerPluginViewerDef
- launchViewer.jsp
- Configuration.html
- MANIFEST.MF

### *The MyViewerPlugin class*

For our example, the MyViewerPlugin class extends the Plugin class, as shown in Example 10-20.

*Example 10-20 MyViewerPlugin.java*

```
package com.mycompany.viewerplugin;

import java.util.Locale;

import com.ibm.ecm.extension.Plugin;
import com.ibm.ecm.extension.PluginService;
import com.ibm.ecm.extension.PluginViewerDef;
import com.ibm.ecm.util.MessageUtil;

public class MyViewerPlugin extends Plugin {
   public String getId() {
      return "MyViewerPlugin";
   }

   public String getName() {
      return getName(Locale.getDefault());
   }

   public String getName(Locale locale) {
      String name = MessageUtil.getMessage(locale,
            "plugins.myViewerPlugin.name");
```

```
        if ((name == null) || (name.equals("plugins.myViewerPlugin.name"))) {
            name = "MyViewer";
        }

        return name;
    }

    public String getVersion() {
        return "1.3";
    }

    public String getDojoModule() {
        return "myViewerPluginDojo";
    }

    public PluginViewerDef[] getViewers() {
        return new PluginViewerDef[] { new MyViewerPluginViewerDef() };
    }

    public PluginService[] getServices() {
        return new PluginService[] { new MyViewerPluginService() };
    }

    public String getConfigurationDijitClass() {
        return "myViewerPluginDojo.ConfigurationPane";
    }
}
```

**Replace the package:** The `com.mycompany.*` package is used here to show that the packages that are used in the SamplePlugin, which are provided with the product, do not have to be followed to successfully develop a plug-in. Most organizations already have a standardized package they use. Replace `com.mycompany` with the package from your organization.

### *The MyViewerPluginService class*

For our example, the MyViewerPluginService class extends the PluginService class, as shown in Example 10-21.

*Example 10-21 MyViewerPluginService.java*

```
package com.mycompany.viewerplugin;

import java.io.IOException;
import java.io.Writer;
import java.net.URLDecoder;
import java.net.URLEncoder;
```

```
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ibm.ecm.P8ParamConstants;
import com.ibm.ecm.extension.PluginService;
import com.ibm.ecm.extension.PluginServiceCallbacks;
import com.ibm.ecm.serviceability.Logger;
import com.ibm.ecm.util.Util;
import com.ibm.ecm.util.p8.P8DocID;
import com.ibm.json.java.JSONObject;

public class MyViewerPluginService extends PluginService {
   public static final String GET_CONFIGURATION = "getConfiguration";
   public static final String SET_CONFIGURATION = "setConfiguration";
   public static final String OPEN_VIEWER = "openViewer";
   private static Set<String> serverTypes = new HashSet();

   public String getId() {
      return "MyViewerService";
   }

   public void execute(PluginServiceCallbacks callbacks,
         HttpServletRequest request, HttpServletResponse response)
         throws Exception {
      String api = request.getParameter("api");
      if (api.equals("getConfiguration"))
         executeGetConfiguration(callbacks, request, response);
      else if (api.equals("setConfiguration"))
         executeSetConfiguration(callbacks, request, response);
      else if (api.equals("openViewer"))
         executeOpenViewer(callbacks, request, response);
   }

   private void executeGetConfiguration(PluginServiceCallbacks callbacks,
         HttpServletRequest request, HttpServletResponse response)
         throws Exception {
      System.out.println("MyViewerPluginService.executeGetConfiguration");

      String configuration = callbacks.loadConfiguration();
      Writer w = response.getWriter();
      w.write(configuration);
      w.flush();
   }

   private void executeSetConfiguration(PluginServiceCallbacks callbacks,
```

```
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {
         String configuration = request.getParameter("configuration");

         if (configuration == null) {
            configuration = "";
         }

         System.out
               .println("MyViewerPluginService.executeGetConfiguration -
configuration: "
                     + configuration);

         callbacks.saveConfiguration(configuration);
      }

      private void executeOpenViewer(PluginServiceCallbacks callbacks,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {
         String serverType = request.getParameter("serverType");

         if (serverTypes.contains(serverType.toLowerCase())) {
            Configuration configuration = new Configuration(
                  callbacks.loadConfiguration());

            if (configuration.configured) {
               StringBuffer urlBuf = new StringBuffer(
                     "VirtualViewer/launchViewer.jsp?");

               String docUrl = request.getParameter("docUrl");

               Map docUrlMap = Util.getParameterMap(docUrl);

               String objectStoreName = (String) docUrlMap
                     .get("objectStoreName");

               String docId = (String) docUrlMap.get("docid");
               docId = (docId != null) ? URLDecoder.decode(docId, "UTF-8")
                     : null;

               String vsId = (String) docUrlMap.get("vsId");
               vsId = (vsId != null) ? URLDecoder.decode(vsId, "UTF-8") : null;

               P8DocID p8DocID = new P8DocID(docId);
               String p8ObjectStore = objectStoreName;
               String reposID = p8DocID.getObjectID();

               String repositoryId = (String) docUrlMap
                     .get(P8ParamConstants.PARM_REPOSITORY_ID);
```

```
            urlBuf.append("p8ObjectStore=").append(
                  URLEncoder.encode(p8ObjectStore, "UTF-8"));
            urlBuf.append("&reposID=").append(
                  URLEncoder.encode(reposID, "UTF-8"));
            urlBuf.append("&vsId=")
                  .append(URLEncoder.encode(vsId, "UTF-8"));
            urlBuf.append("&repositoryId=").append(
                  URLEncoder.encode(repositoryId, "UTF-8"));
            urlBuf.append("&docid=").append(
                  URLEncoder.encode(docId, "UTF-8"));

            Logger.logInfo(MyViewerPluginService.class,
                  "executeOpenViewer", request.getSession(),
                  urlBuf.toString());

            response.sendRedirect(urlBuf.toString());
         } else {
            throw new Exception("MyViewer has not been configured.");
         }
      } else {
         throw new Exception("Unsupported server type: " + serverType);
      }
   }

   static {
      serverTypes.add("p8");
   }

   private class Configuration {
      boolean configured = false;
      boolean annotationsEnabled;

      public Configuration(String configuration) throws IOException {
         JSONObject jsonConfig;

         System.out.println("Configuration.ctor - configuration: "
               + configuration);

         if (configuration != null) {
            jsonConfig = JSONObject.parse(configuration);

            String annotationsEnabled = ((String) jsonConfig
                  .get("annotationsEnabled"));

            this.configured = (annotationsEnabled != null);

            this.annotationsEnabled = Boolean
                  .parseBoolean(annotationsEnabled);
```

```
         }
      }

      public String serialize() throws IOException {
         JSONObject jsonConfig = new JSONObject();

         System.out.println("Configuration.serialize -
this.annotationsEnabled: "
               + this.annotationsEnabled);

         jsonConfig.put("annotationsEnabled", this.annotationsEnabled);

         return jsonConfig.serialize();
      }
   }
}
```

### *The MyViewerPluginViewerDef class*

For our example, the MyViewerPluginViewerDef class extends the PluginviewerDef class, as shown in Example 10-22.

*Example 10-22 MyViewerPluginViewerDef.java*

```
package com.mycompany.viewerplugin;

import com.ibm.ecm.extension.PluginViewerDef;
import java.util.Locale;

public class MyViewerPluginViewerDef extends PluginViewerDef {
   private static final String[] SUPPORTED_SERVER_TYPES = { "p8" };

   private static final String[] SUPPORTED_MIME_TYPES = { "application/pdf" };

   public String getId() {
      return "myViewer";
   }

   public String getName(Locale locale) {
      return "MyViewer";
   }

   public String getLaunchUrlPattern() {
      return
"servicesUrl+'/plugin?plugin=MyViewerPlugin&action=MyViewerService&api=openView
er&docUrl='+encodeURIComponent(docUrl)+'&contentType='+mimeType+'&serverType='+
serverType+privs";
   }
```

```
    public String[] getSupportedContentTypes() {
        return SUPPORTED_MIME_TYPES;
    }

    public String[] getSupportedServerTypes() {
        return SUPPORTED_SERVER_TYPES;
    }
}
```

### *The launchViewer.jsp file*

For our example, we also need a JSP file to load and redirect to our viewer. We can add it anywhere in the project and name it `launchViewer.jsp` file. For this example, we place it under the `VirtualViewer` directory. Example 10-23 shows the JSP file.

*Example 10-23 JSP file which redirects to the viewer (launchViewer.jsp)*

```
<%@ page import="java.net.URLDecoder,java.net.URLEncoder"%>
<%
    request.setCharacterEncoding("UTF-8");

    String clientInstanceId = request.getSession().getId();

    String objectStoreName = request.getParameter("p8ObjectStore");

    objectStoreName = (objectStoreName != null ?
URLDecoder.decode(objectStoreName, "UTF-8") : null);

    String vsId = request.getParameter("vsId");

    vsId = (vsId != null ? URLDecoder.decode(vsId, "UTF-8") : null);

    String repositoryId = request.getParameter("repositoryId");

    repositoryId = (repositoryId != null ? URLDecoder.decode(repositoryId,
"UTF-8") : null);

    String docId = request.getParameter("docid");

    docId = (docId != null ? URLDecoder.decode(docId, "UTF-8") : null);

    String documentId = "objectStoreName=" + objectStoreName
           + "&folderId=" + "null" + "&objectType=" + "document"
           + "&vsId=" + vsId + "&id=" + docId + "&repositoryId="
           + repositoryId;

    String encodedDocumentId = URLEncoder.encode(documentId);
%>
```

```
<html>
<head>
<title>Redirecting to Viewer</title>
<meta http-equiv="REFRESH"
content="0;url=/navigator/VirtualViewer/ajaxClient.html?documentId=<%=encodedDo
cumentId%>&clientInstanceId=<%=clientInstanceId%>">
</head>
</html>
```

### *The Configuration.html file*

We use the `Configuration.html` file, from "The ConfigurationPane.html file" on page 382, and add it in the following package:

`com.mycompany.viewerplugin.WebContent.myViewerPluginDojo.templates`

### *The Configuration.js file*

We use the `Configuration.js` file, from "The ConfigurationPane.js file" on page 382, and add it in the following package:

`com.mycompany.viewerplugin.WebContent.myViewerPluginDojo`

### *The MANIFEST.MF file*

We use the `MANIFEST.MF` file, from "The MANIFEST.MF file" on page 387, and add it to the root of the project. We make sure this manifest file is used during the creation of the JAR file (the `build.xml` file takes care of that for you; the `build.xml` is shown in "Deploying and setting up the plug-in" on page 397).

## Deploying and setting up the plug-in

**Important deployment strategy:** This tutorial places many files in the IBM Content Navigator deployed path. Redeploying the product might result in these files being deleted. Be sure you have a proper deployment strategy for your plug-in.

To deploy the plug-in, complete the following steps:

1. Build a simple ANT file to produce a JAR file. Our build file is shown in Example 10-24. This build file compiles the code and builds the JAR file in a folder named `build`. This file can be run directly in Eclipse or from a command line.

*Example 10-24 Build file to deploy our plug-in (build.xml)*

```
<?xml version='1.0' encoding='UTF-8'?>
<project
```

```
name="myViewerPlugin"
basedir="."
default="all" >

<target
    name="all"
    depends="clean,compile,jar" />

<path id="classpath" >

    <fileset
        id="lib.jars"
        dir="lib" >

        <include name="*.jar" />
    </fileset>
</path>

<target name="clean" >

    <delete dir="build/temp" />
</target>

<target name="compile" >

    <mkdir dir="build/temp" />

    <javac
        debug="true"
        destdir="build/temp"
        source="1.5"
        srcdir="src"
        target="1.5" >

        <classpath refid="classpath" />

        <include name="**/*.java" />
    </javac>
</target>

<target name="jar" >

    <copy todir="build/temp" >

        <fileset dir="src" >

            <include name="**/WebContent/**" />
        </fileset>
    </copy>
```

```
        <jar jarfile="build/MyCompanyViewer.jar" >

            <fileset
                dir="./build/temp"
                includes="**/*" />

            <manifest>

                <attribute
                    name="Plugin-Class"
                    value="com.mycompany.viewerplugin.MyViewerPlugin" />

                <section name="build" >

                    <attribute
                        name="Built-By"
                        value="${user.name}" />

                    <attribute
                        name="Build"
                        value="${TODAY}" />
                </section>
            </manifest>
        </jar>

        <delete dir="./build/temp" />
    </target>

</project>
```

2. After the project is built, deploy your solution. For our example, we use the following steps:
   - Place the JAR file in the plug-ins directory. In our setup, we use the following path:

     `C:\<WAS_INSTALL>\AppServer\profiles\AppSrv01\installedApps\P8Node01Cell\navigator.ear\navigator.war\plugins`
   - Place `launchViewer.jsp` file in the VirtualViewer subdirectory. In our setup, we use the following path:

     `C:\<WAS_INSTALL>\AppServer\profiles\AppSrv01\installedApps\P8Node01Cell\navigator.ear\navigator.war\VirtualViewer`

3. After deployment is complete, register the new plug-in. We use the following steps, in the IBM Content Navigator Administration Desktop:
   a. Open the **Plugins** tab.
   b. Specify the file by using a URL that points to the deployed JAR file. We use the following URL:

      `http://<ip/host>/navigator/plugins/MyCompanyViewerPlugin.jar`

      The plug-in registration page is shown in Figure 10-9.

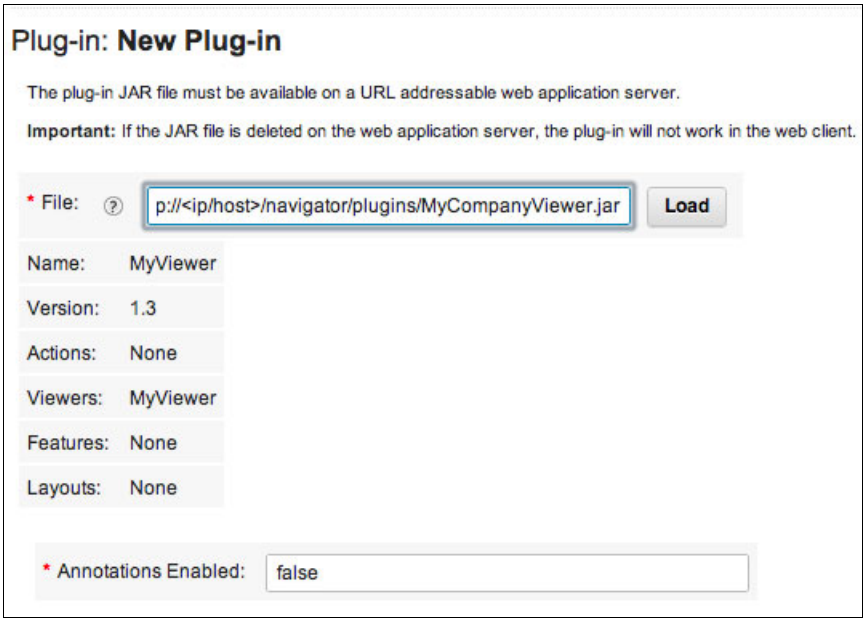



*Figure 10-9 Plug-in registration page*

You can use a more descriptive name for the JAR file. For our example, we use `SnowboundViewerPlugin.jar` file. However, we have a distinction between what is made available by Snowbound and what must be built by us to complete the example. We do not need to specify any configuration; the *annotationsEnabled* configuration setting is an example of how to build the configuration details.

c. Click **Save and Close**. The plug-in is now registered and is listed with the others, as Figure 10-10 shows.

*Figure 10-10 List of registered plug-ins*

4. Associate one or more MIME types with the new viewer so IBM Content Navigator can invoke it when a user clicks to open a document. Do that in one of two ways:
   - Create a viewer map and set up the MIME types there.
   - Modify the map that is already associated with the desktop you use and add the new MIME type mapping there.

   For our example, we use the second way. The map that is defined for the desktop we use is named My Viewer Map. We then use the following steps to associate the new MIME types with the new viewer:

   a. Go to **Viewer Maps** in the Administration Desktop.

   b. Click **New Mapping** to this viewer map.

   c. Set up the supported configurations. The IBM Content Navigator user interface already knows what MIME types and repository types the new viewer supports, so it guides you with setting up only the supported configurations.

Our specific setup is shown in Figure 10-11.

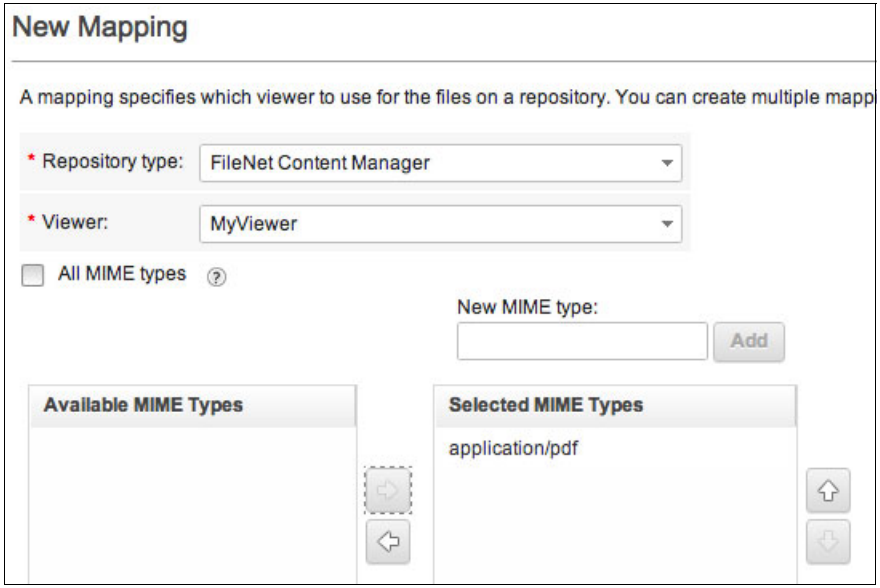



*Figure 10-11 The viewer mapping configuration page*

d. Be sure that the precedence rules in the viewer map do not override your configuration. For example, if another viewer is defined with support for PDFs and it is listed before your configured viewer, IBM Content Navigator uses the first viewer that is listed.

For our example, we click **Move up** (Figure 10-12) to be sure that our viewer takes precedence over all others. That means our viewer is first in the list of mappings.

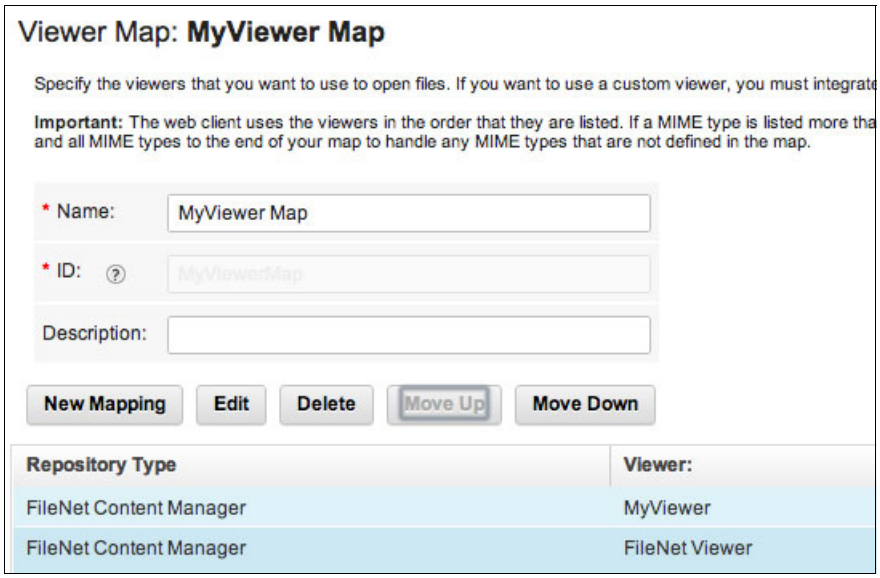



*Figure 10-12 Observing viewer map precedence rules*

e. Before testing the viewer, refresh the browser to make sure that the changes are in effect.

5. To test your viewer, go to your repository, locate and open the document. For our example, we go to a IBM FileNet P8 repository, find a PDF document, and click to open it.

The Snowbound VirtualViewer is invoked and renders the document, as shown in Figure 10-13.

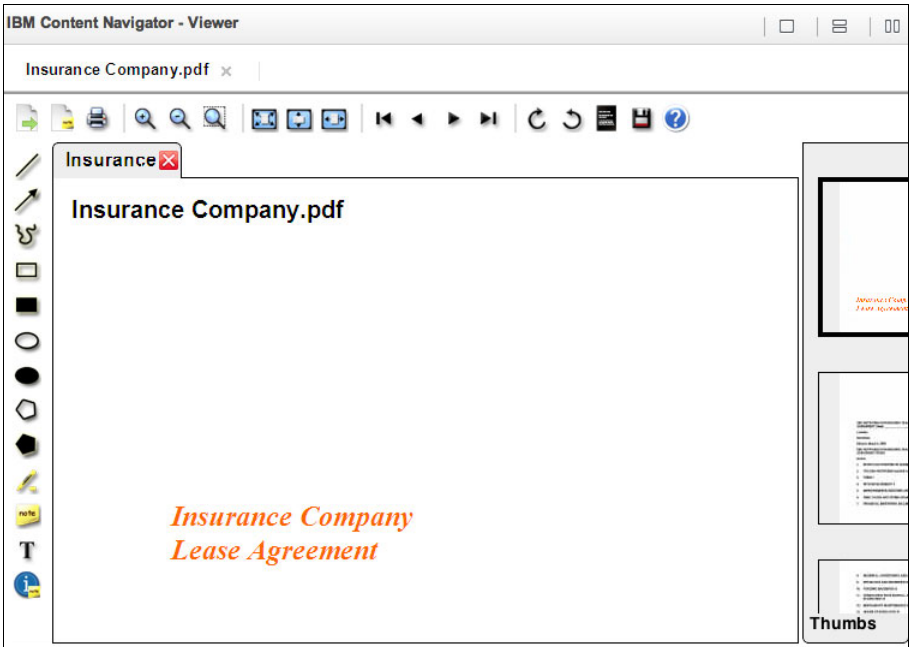



*Figure 10-13 The viewer has been launched*

## Using the Snowbound VirtualViewer ICN Connector

To bypass the need to develop your own plug-in to call the VirtualViewer, you can use the available plug-in package provided by Snowbound. The plug-in we use here is the VirtualViewer ICN Connector. We downloaded the evaluation version of this plug-in, which is available from the Snowbound Software website or though the following location:

`http://downloads.snowbound.com/evaluate/73jGOCvaZ910k2/vvRedbook.php`

## Registering the plug-In

The Snowbound VirtualViewer plug-in must be registered as a plug-in within the Content Navigator environment. The VirtualViewer plug-in is named `SnowboundVirtualViewerPlugin.jar` and you must make this file available to the Content Navigator environment.

To register the plug-in, open the Content Navigator Administration Desktop:

1. Select **Plug-ins** from the configuration options and select **New plug-in**.
2. Select the JAR file option and enter the location for the `SnowboundVirtualViewerPlugin.jar` file.
3. Click **Load**. If all is correct, the window auto-completes with configuration options that are specific to the VirtualViewer plug-in. See Figure 10-16 on page 409.
4. Enter the location of the VirtualViewerURL that you configured as part of the prerequisites:

   `/navigator/VirtualViewerHTML5`
5. Enter the location of the VirtualViewer servlet path:

   `/navigator/AJAXServlet`
6. Save and close the plug-in configuration.

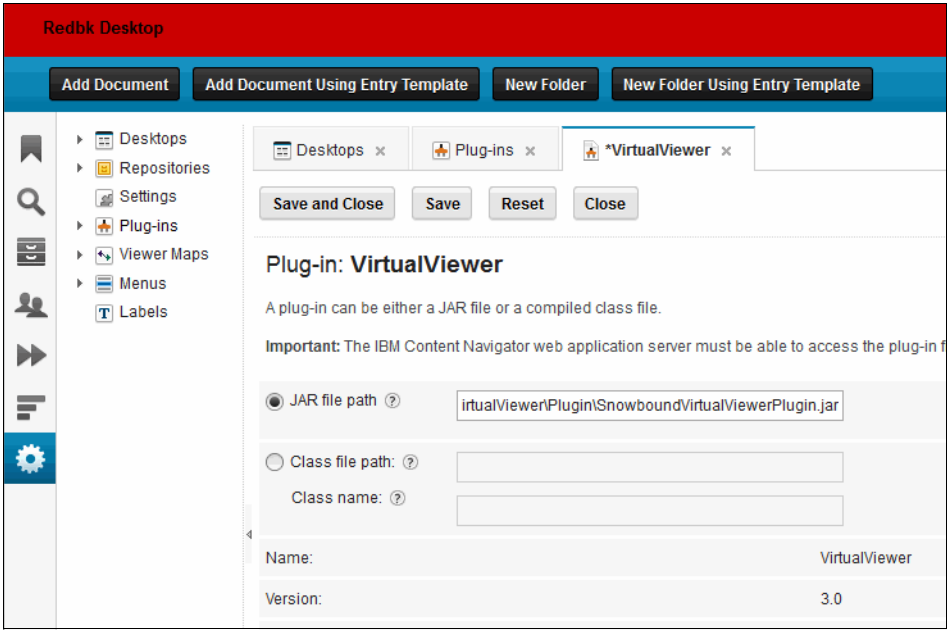



*Figure 10-14 Registering the Snowbound VirtualViewer Plug-in*

## Configure virtual viewer in the IBM Content Navigator desktop

After the plug-in is registered, you can use the plug-in as part of a Content Navigator desktop configuration. Several steps are required to enable the plug-in with a desktop configuration:

1. Configure a viewer map.

   The viewer map configuration determines the association between a specific MIME type or types and an identified viewer. With this features, you can determine which viewer the system will launch based on the associated file type. In this way, you can predetermine which file types should be launched with the VirtualViewer.

   Viewer maps are configured within the Administrator Desktop as follows:

   a. Select **Viewer Maps** from the configuration options and select **New viewer map**.

   b. Name the viewer map configuration.

   c. Click **Repository Type** and select **FileNet Content Manager**.

   d. From the drop-down menu next to Viewer, select **VirtualViewer**.

   e. Select the MIME types you want to associate with this viewer map configuration.

   Figure 10-15 on page 407 shows our set up for the Snowbound VirtualViewer MIME type mapping.

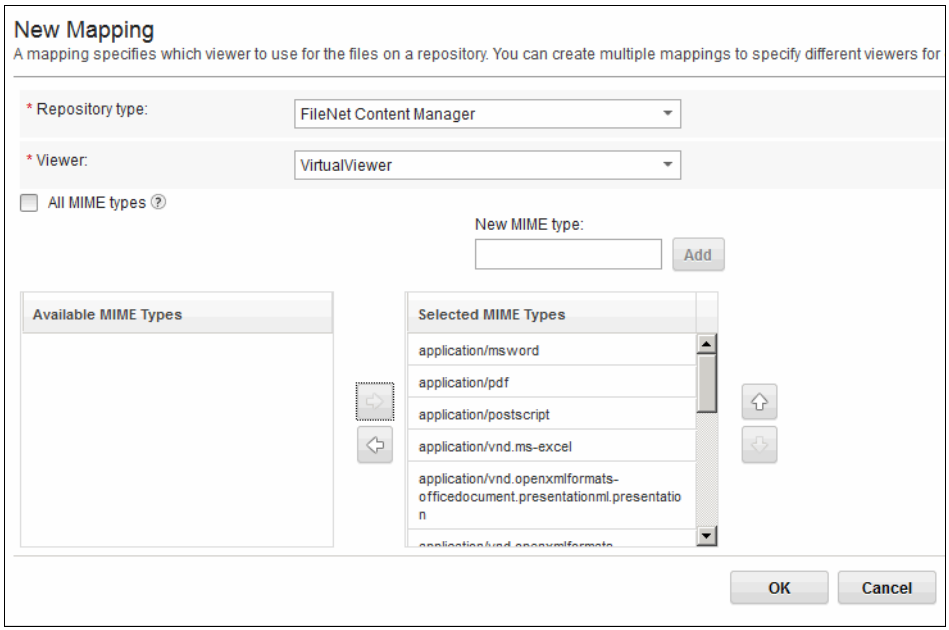



*Figure 10-15 Viewer MIME type mapping*

2. Configure a desktop to use the viewer map:
   a. In the Desktops settings, select a desktop where you will use your newly created viewer map.
   b. Set **Viewer Map** to the newly created **VirtualViewer HTML5 Map**.
3. Restart the server running your web server to assure that VirtualViewer is functioning correctly with the IBM Content Navigator environment.

## 10.5.2 Informative Graphics Brava! Enterprise Viewer

The Brava! Plug-in for IBM Content Navigator leverages the plug-in architecture of the Navigator environment. Brava provides complimentary viewing, annotation, redaction, and publishing features from within the IBM Content Navigator experience.

Supplementing your IBM Content Navigator with Brava! allows you to extend the capabilities of the navigator environment to include support for additional file formats, document merging capabilities, supplemental search support, and many other operations.

The Brava! Viewer plug-in delivers the option to use an HTML-based or an Active-X based viewer, depending upon your organization's requirements and support environment.

In this example, we configure IBM Content Navigator to use the Brava! Viewer to provide document-compare and document-merge capabilities. The document-merge operation allows Content Navigator users to select one or more documents or document versions within the navigator desktop and rearrange the pages within the Brava! Viewer or create a new document. The document-compare command will allow users to compare either two versions of the same document, or two different documents at the same time.

## Prerequisites

Before we start, certain preparation work must be completed. The Brava Enterprise Administrator's Guide has instructions for installing and configuring the software. Before you begin configuring the IBM Content Navigator client to use the Brava! Viewer, the entire installation and deployment process must be completed.

The `brava.properties` file must be located on the Content Navigator Server and it must be updated to include your local Brava! properties:

- `server.brava`: Should include the URL to the location of the Brava Server
- `application.url`: Should refer to the URL that clients use to access the Content Navigator Application
- `server.viewer.files`: Should include an accurate path to the client installation files
- `compare.client`: Should indicate which viewer to launch for the document compare operation (HTML or activeX)

## Registering the Plug-In

The Brava! Viewer plug-in must be registered as a plug-in within the Content Navigator environment. The Brava! Viewer includes a Content Navigator plug-in named IGCPlugin. You must deploy this plug-in either through a web server or drive that is accessible to the Content Navigator environment. To register the plug-in, open the Content Navigator Administration Desktop and complete the following steps:

1. Select **Plug-ins** from the configuration options and select **New plug-in**.
2. Select the JAR file option and enter the location for the `IGCPlugin.jar` file.
3. Click **Load**. If this is done correctly, the window auto-completes with the configuration options that are specific to the Brava! plug-in. See Figure 10-16 on page 409.

4. Enter the location of the `brava.properties` file that you configured as part of the prerequisites. Remember, this file must be local to the Content Navigator server.
5. Save the plug-in configuration.

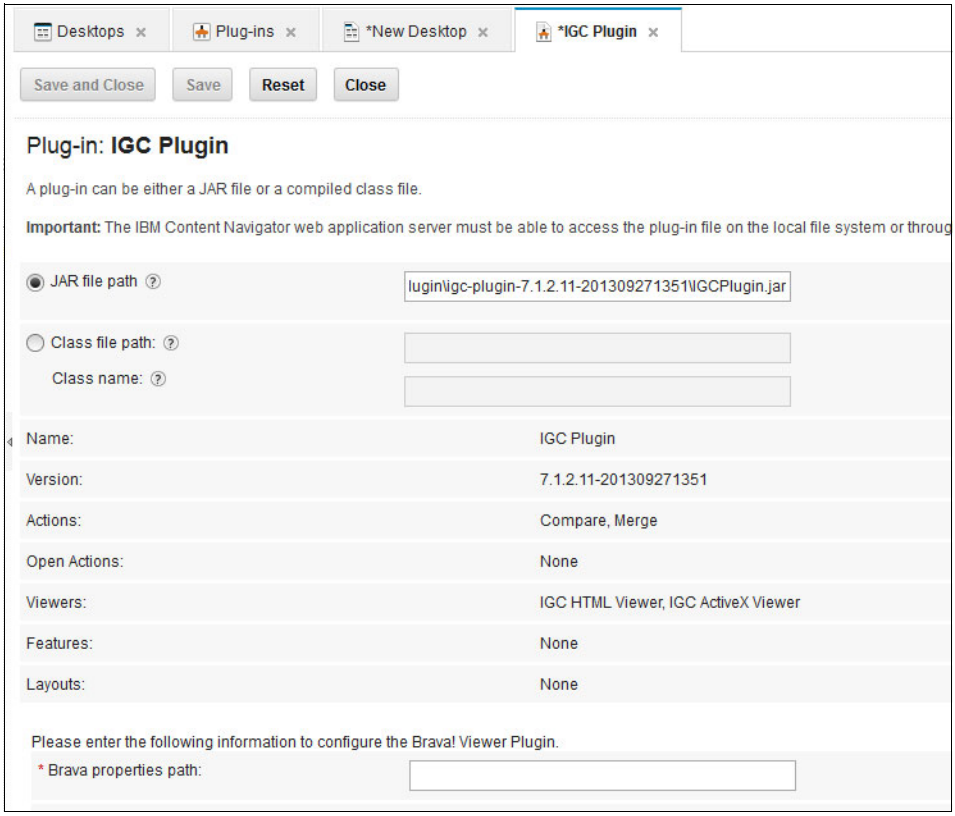



*Figure 10-16 Configuring the IGC Viewer Plug-in*

## Configuring the IBM Content Navigator Desktop

After the plug-in is registered, you may now use it as part of a Content Navigator desktop configuration. Several steps are required to enable the plug-in with a desktop configuration:

1. Configure a viewer map.

   The viewer map configuration determines the association between a specific MIME type or types and an identified viewer. With this feature, you can determine which viewer the system will launch based on the associated file type.

In this way, you can pre-determine which file types should be launched with the Brava! Viewer.

Configure viewer maps within the Administrator Desktop as follows:

a. Select **Viewer Maps** from the configuration options and select **New viewer map**.

b. Name the viewer map configuration.

c. Click **New mapping** and select your repository type (IBM Content Manager or FileNet Content Manager).

d. Select either HTML or ActiveX. At the time of writing this book, the HTML viewer is certified on Internet Explorer 9 (Windows), Firefox (Mac and Windows), and Safari (Mac and Windows). The ActiveX Viewer is certified on Internet Explorer 8 and 9 (Windows).

e. Select the MIME types you want to associate with this viewer map configuration.

Figure 10-17 shows our set up for the IGC ActiveX Viewer.

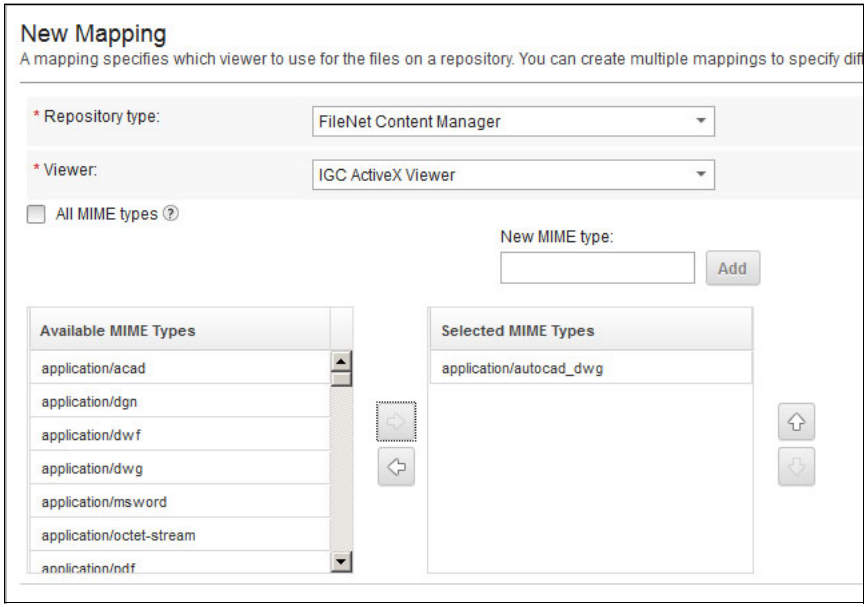



*Figure 10-17 Viewer Mapping*

2. Configure a context menu.

   By configuring the content menu, you can add specific Brava! Viewer functions to the default application menu. In this case, we add the Compare and DocMerge functions to the menu.

Configure context menus in the Administrator Desktop as follows:

a. Select **Menus** from the configuration options.

b. Make a copy of the Default Document Context menu and rename it.

c. Select the **Compare** and **DocMerge** actions from the list of available actions and place them in the location you want within the menu list by using the up and down arrows.

3. Configure a Navigator Desktop to use the viewer map and menu.

   Adding the viewer map and the menu to a desktop causes those items to be visible within a specified Content Navigator Desktop.

   Configure Navigator Desktop in the Administrator Desktop as follows:

   a. Select **Desktops** from the configuration options.

   b. Select the desktop configuration with which you want to use the Brava! Viewer.

   c. Click **Edit**.

   d. In the **General** tab, select the viewer map you have just configured in the Desktop Configuration section.

   e. In the **Menus** tab, select the menu you just configured in the Document content menu field in the Context Menus section.

4. Install the IGC Extensions AddOn in IBM FileNet Content Manager.

   Within the FileNet Enterprise Manager administration application, install the IGC Extension to the environment:

   a. Within FileNet Enterprise Manager, right-click **AddOns** and select **New AddOn**.

   b. Select the **IGC_CoreExtensions.xml** file from the Brava! Enterprise installation archive.

   c. Click **OK** and then add the AddOn to the object stores you want.

   d. Select the object store you want, and then right-click and select **All Tasks** → **Install AddOn**.

   e. Select the AddOn you imported to the environment.

## 10.6 Conclusion

This chapter has an overview of the built-in viewers within IBM Content Navigator. If your customization needs are not met by the built-in viewer functions, you can extend viewing capabilities through client-side or server-side extension development through the Navigator API.

In addition, various third-party viewers provide enhanced viewing capabilities such as merging and comparing documents and extended format viewing. This chapter shows how two of the third-party viewers can be easily configured for use through Content Navigator plug-in architecture.

11

# Extending solutions to mobile platform

This chapter describes scenarios that can be used for extending a solution based on IBM Content Navigator to a mobile platform. It provides an introduction to an IBM Worklight sample that is included with the product and demonstrates customization of this sample. In addition, this chapter describes deployment scenarios that can be used for this sample.

This chapter covers the following topics:

- IBM Content Navigator mobile solutions
- Example overview
- Customizing IBM Worklight sample

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# 11.1 IBM Content Navigator mobile solutions

Mobile access to corporate content is becoming a real need. This need is no longer limited to knowledge workers such as claim adjusters who are familiar with mobile applications and are on constant move. Companies are under pressure to use a mobile platform for their employees and clients to increase work efficiency, accelerate information distribution and gain. Today, ECM systems are being exposed to the public Internet, to have content directly accessed and used by external customers, and accessed at any time and from any place. This new approach helps to avoid replication or printing of the documents and ensures users always have necessary documents of their latest versions. Furthermore, they can leverage more ECM features like metadata or content-based searches. There is no need to confine this only to content-based features; mobile applications can open up process capabilities, allowing users to participate in workflow and work with data integrated from ECM and external systems.

In a scenario where a solution is built as an extension to IBM Content Navigator, advantages can be gained from modularity that its architecture provides. It is common that a mobile client provides a subset of functionalities that are normally offered by a standard web client. Developing mobile applications using the IBM Content Navigator JavaScript model simplifies reuse of existing components. All extensions of server-side components such as EDS, request/response filters, and service plug-ins, can be also used directly from the mobile client.

## 11.1.1 Mobile development options

Mobile devices are specific in the way that users interact with a web page. Control through touch screens requires larger areas where the user can touch to better address differences between accuracy of a computer mouse pointer and a human finger. Instead of a left and right mouse click, the distinguishing features are swipe, tap, pinch, or device rotation. Devices also differ in resolution and provide easier access to functions like zoom, camera, gyroscope, or detection of a user's position using GPS. Speed of Internet access is often limited by available type of network or WiFi. While developing mobile applications, it is important to decide what level of mobile specifics you want to adopt in your solution.

**Note:** This section does not include all possible options and focuses only on those that are most frequently used.

## Mobile web application

To create a web application similar to the native apps, you can use one of the web frameworks that have support for a mobile platform and imitate its look and feel. One such frameworks is Dojo Mobile:

http://dojotoolkit.org/features/mobile

This framework is an extension of Dojo framework and thus simplifies integration with IBM Content Navigator that uses the same base. This framework provides a large suite of widgets offering a good user experience and supporting easy control using finger movements such as touch, scroll, or tap. You can also mix Dojo mobile widgets with standard dijits provided by IBM Content Navigator. This approach allows you to reuse developed UI elements you might already have for your standard web application or only those that are included with the product. By using the IBM Content Navigator JavaScript you can also call your custom service plug-ins. This combination brings you a high level of component reuse.

It is important to note that Dojo Mobile still produces HTML5 and Java Script web application that has limited access to local devices. With HTML5, development of new specifications is underway. An example includes getUserMedia/Stream API allowing you to work with external device data such as a camera. They might not, however, be fully supported by all mobile browsers. See “Hybrid mobile application” on page 416 if you want to access local devices.

## Mobile optimized layout of IBM Content Navigator

Another important note is that IBM Content Navigator is written using HTML5 and can be directly displayed in modern browsers on mobile devices. Comfort of the user experience grows with the width of the device screen. You can further improve this by creating multi-channel layout or dijits. Dojo also provides *dojo/has* and *dojo/sniff* modules that you can use for user agent tests. In your layout or custom features, you can change the organization of dijits based on the target platform. You can also use conditional module loading based on the client environment. Dojo loader supports ternary expression, which you can use for conditional module loading, enabling you to reduce network traffic from the mobile device.

## Extension of native iOS client

IBM Content Navigator is included with the native iOS client that can be downloaded from the Apple AppStore. The user interface of the native client fully exploits the potential of the native application and provides a strong user experience. This client is suitable in a scenario when you need to provide a standard set of functions to mobile users. This client communicates with midtier services, which makes it partially possible to customize the behavior of the standard features by using response/request filters. In addition, this client also has low demand on network bandwidth because it transfers only data, without UI elements. This is not true for another customization option that the iOS client provides where you can add custom features that open a specified URL directly within the native client. This way you can easily integrate the native client with other web applications. It is also possible to open your IBM Content Navigator based applications using this second option. You can benefit from sharing session cookies with the native client in case your application resides on the same server as the desktop you opened from the native client and the user will not be required to log in again.

## Hybrid mobile application

Native application provides high performance and access to device APIs, and reduces network demand but requires specific knowledge of the development platform. However, web applications provide a certain level of portability and simplification of development but with limited access to device APIs. Combining the benefits of both approaches is called a *hybrid application* and consists of these parts:

- Native part: This is a native application deployed directly on the device. It provides an extended web browser and container where you can store any of your content. You can store your web content as HTML, JavaScript, and images directly on the device without the need for repetitive downloading or reliance on cache. This accelerates speed of the web application. *Apache Cordova* is one of such frameworks that provides native shell-wrapping web content. It also provides a unified cross-platform Java Script API that can be used to access a physical device API without needing to write any native code and understanding platform specifics.
- Web part: This is the web content displayed by the native part. There is no limit in what the web application can use because it runs within web browser. You can use *Dojo Mobile* or any other framework you need. Additionally, you can use a JavaScript API described in the previous bullet.

We mentioned several frameworks you can use for your development. For enterprise-level development, you can use *IBM Worklight* that provides strong tooling for hybrid, web, and native mobile development. It uses the frameworks we mentioned and brings additional features combined into a single environment.

You can use the IDE for development and testing, which provides features for debugging, integration with device SDK, single build for multiple target platforms, JavaScript API for native access, and a browser-based simulator. With this approach, you can generate a web or hybrid application than does not need any IBM Worklight runtime infrastructure for production use. However it can still benefit from use of IBM Worklight APIs.

You can also extend your application to comprise enterprise-level features such as managed deployment to target devices, version management, intelligent analytics tracking of user and application behavior, custom event and audit logging, push notifications, or a wide range of adapters. For such scenarios, you will also need additional infrastructure such as the Worklight Server. See the product documentation for more details:

http://pic.dhe.ibm.com/infocenter/wrklight/v6r0m0/topic/com.ibm.help.doc/wl_home.html

## 11.2 Example overview

Next, we introduce a mobile sample provided with IBM Content Navigator and demonstrate customization with IBM Worklight and Dojo mobile.

IBM Content Navigator provides a mobile sample project written in IBM Worklight Studio. This sample demonstrates the use of standard content management capabilities like favorites, search, Datacap, or a repository browser that is optimized for an iPhone client. The project contains source code that you can import to your environment and use as a base for your own mobile solution.

Before customization, be sure you understand at least the high-level architecture of the sample and deployment scenarios. We focus on deployment of a hybrid application as shown in Figure 11-1 on page 418.

Later, in our customization example, we show another way of packaging the sample: it will be packaged and deployed as an IBM Content Navigator plug-in directly on a server. See 11.3.7, “Packaging and deployment” on page 432 for further details.

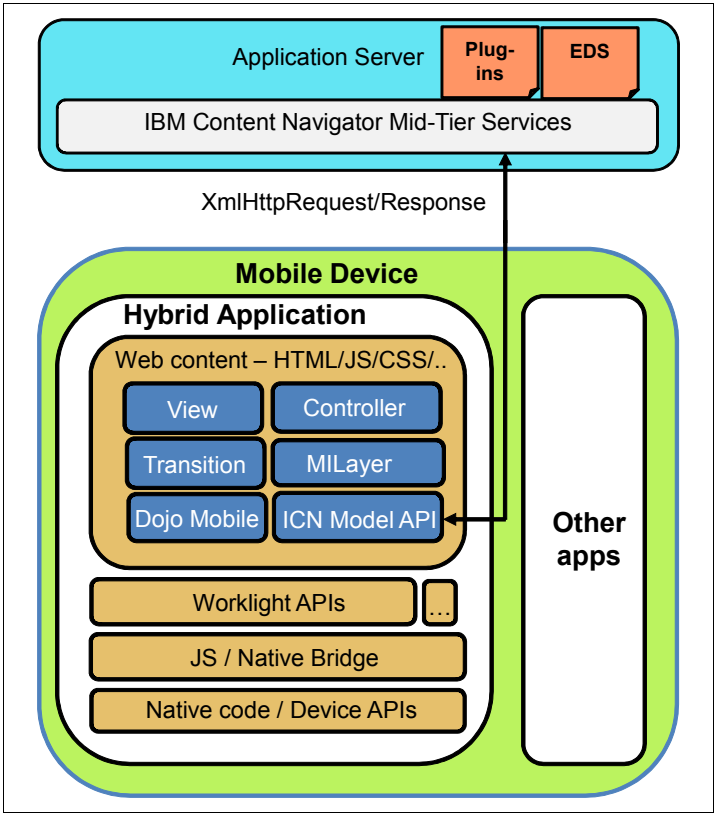



*Figure 11-1 Architecture of mobile sample*

When packaging as a hybrid application, the web content on the device will be displayed within its native shell. Although most of the web content is loaded from the local device, the hybrid application still relies on the server-side for model-related operations. It uses IBM Content Navigator Model API that communicates with midtier services. The sample uses a *model-view-controller* pattern. The *model* API is not accessed directly from the application but uses a singleton facade layer referred to as *MILayer*.

The whole user interface, including presentation data model, is realized using Dojo Mobile widgets. To work with *views*, it is possible to use support APIs. These include for example a *transition* API that performs a segue between views or a progress indicator that informs the user about active operations that might require some time to process.

The *controller* prepares a view based on model data. In most cases, it also handles user events based on which it updates. It also interacts with Worklight JavaScript APIs (for example to access camera or photo albums using Cordova API). To use advanced features like document preview or interaction with other applications, the controller can invoke parts of the native code.

If the sample detects that it is being run as a hybrid application it will use an IBM Worklight API (*JSONStore*) to store the application password. This store can be AES-256 encrypted.

With sufficient understanding of the architecture, we focus on customization.

## 11.3 Customizing IBM Worklight sample

This section describes the process for adding a new Work feature to the IBM Worklight sample.

**Shorthand notation:** We refer to the following path as the `<common folder>`:

`mobileHybrid/apps/mobileHybridApp/common`

### 11.3.1 Example overview

To illustrate customizability, we extend this sample to provide a new Work feature as shown on Figure 11-2 on page 420. IBM Content Navigator allows you to configure mobile features used within your desktop. We follow this option and allow the desktop administrators to decide if they want to display this feature for a particular desktop. Despite the differences between IBM FileNet P8 and IBM Content Manager workflow capabilities in our use case, IBM Content Navigator provides a level of abstraction that allows us to support both of them with minimal changes. One of the differences is another hierarchy of work item containers.

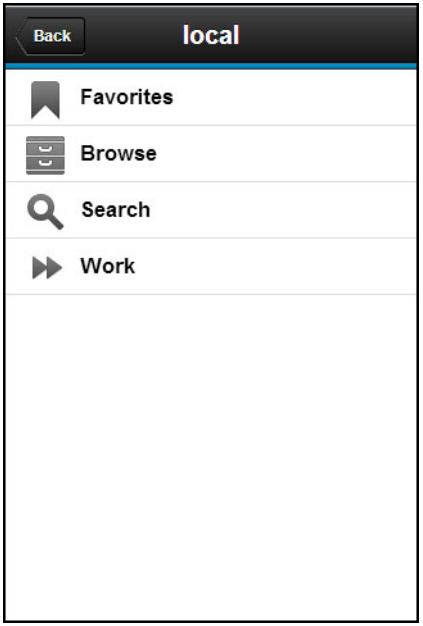



*Figure 11-2 Custom work feature*

In our sample, we find the first suitable work items container and display these work items in the form of a UI, optimized for iPhone resolution as shown on Figure 11-3 on page 421.

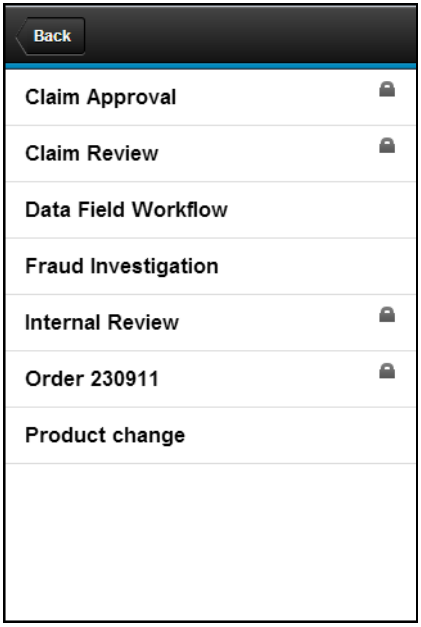



*Figure 11-3 Work item list*

Work items can be locked by the user, indicating that the user is working on them. In this case, we display a lock icon to the right of the work item. Because our new feature is read-only and does not allow users to modify work items, our example does not lock any work items. However, we will not limit the display of the locked items.

You can select any work item and get a list of all defined attributes, as shown on Figure 11-4 on page 422.

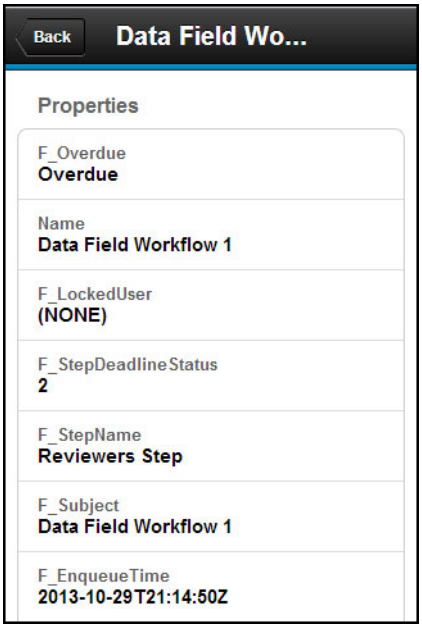



*Figure 11-4 Work item properties*

In a standard scenario, you see that a step processor is assigned to the step. This requires much additional customization that is beyond scope of this book. Remember the purpose of this example is to demonstrate how you can customize the sample and not how to add full workflow support.

## 11.3.2 Environment preparation

The following steps are necessary to prepare the environment for customization:

1. “Configuring IBM Content Navigator” on page 423
2. “Preparing the development environment” on page 424
3. “Packaging and deployment” on page 432

## Configuring IBM Content Navigator

Because we are adding a workflow-related feature, we must prepare the repository with workflow support and at least one *workbasket*.

For this sample, we prepare a new desktop named `SampleMobileDesktop` with the following configuration:

- General tab:
  - Name and ID: SampleMobileDesktop
  - Authentication: Select prepared repository
- Mobile tab:
  - Mobile application access: Enabled
  - In the Features section, select **New Feature** and enter this information:
    - Name: Work
    - Icon File: Provide any address to the icon you want to display in case the desktop will be accessed from a native iOS client.
    - URL: Provide any address you want to display in case the desktop will be accessed from native iOS client. URL address used here does not have any impact on our sample.
  - Confirm by clicking **OK** and then be sure to select the **Display** check box for the new feature, as shown on Figure 11-5. Otherwise the sample ignores this feature.

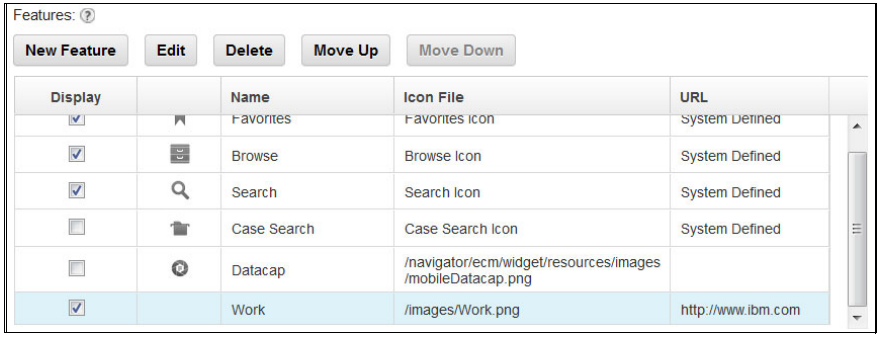



*Figure 11-5 Mobile features configuration*

- Repositories tab:
  - Select the repository with workflow support, as described previously.

- Appearance tab:
  - Default repository: Select the repository you have just assigned to the desktop.
- Workflows tab (IBM FileNet P8 systems only):
  - Repository: Select the repository you have just assigned to the desktop.
  - Application space: Select at least one application space.

Click **Save** and **Close** to save your new desktop configuration.

## Preparing the development environment

Now, prepare the development environment that is used later in this chapter. You need `sampleMobileApp.zip` and `SampleMobilePlugin.jar` files, which are in your `ECMClient_installdir\samples` IBM Content Navigator directory.

**Note:** Be sure to use at least IBM Content Navigator 2.0.2 fix pack 1.

Follow the instructions in the product documentation, which describes how to set up the development environment. Download the instructions from this web page:

`http://www.ibm.com/support/docview.wss?uid=swg27038925`

While configuring the `daddress` property, also include the identifier of the new desktop created previously, as shown in Example 11-1.

*Example 11-1 Desktop configuration*

```
daddress: "http://<ip>:<port>/navigator/?desktop=SampleMobileDesktop",
```

After following the steps in the documentation, you can build the sample and run it within the simulator. The development environment is now set up and ready for development.

**Note:** When connecting from your development environment to IBM Content Navigator server you might violate the *same-origin-policy*. The JavaScript is loaded from a separate location where it is making network calls. The browser can be configured to disable these security checks to allow development and testing, *otherwise the sample will not work in web simulator*. See the browser documentation for further details. In a production environment, you have two options: Deploy as a hybrid application to the device or deploy on the server that is running IBM Content Navigator and run the sample from there.

### 11.3.3 Updating model layer

In this section, we update the facade layer of the sample using the IBM Content Navigator Model API. We add the retrieveWorkItems method that retrieves the first available workbasket containing work items. The hierarchy of work items containers differs in FileNet P8 and IBM Content Manager thus we use the _retrieveWorkItems recursive function to find the first available.

We use these steps:

1. Open the Eclipse project, prepared in “Preparing the development environment” on page 424
2. Open the <common folder>/mila/MILayer.js file and add new methods as listed in Example 11-2.

*Example 11-2 Retrieving work items*

```
_retrieveWorkItems:function(item,finalCallback) {
      if(item.isInstanceOf && (item.isInstanceOf(ecm.model.WorklistFolder)
/* Is container?*/
|| item.isInstanceOf(ecm.model.ProcessRole) ||
item.isInstanceOf(ecm.model.ProcessApplicationSpace))) {
      item.retrieveWorklists(dojo.hitch(this,function(wBasketList){
         item = wBasketList[0];
         this._retrieveWorkItems(item,finalCallback);
      }));
   } else {
      item.retrieveWorkItems(dojo.hitch(this, function(wItems) {
      if(finalCallback) {
         finalCallback(wItems);
      }}),null, null, true,null);
   }
   return true;
},
retrieveWorkItems:function(callback) {
   var repository = this.getCurrentRepository();
   if (repository == null) {
      return false;
   }
   repository.retrieveWorklistContainers(dojo.hitch(this,
function(wlContainers) {
      item = wlContainers[0];
      this._retrieveWorkItems(item,callback);
   }));
         return true;
},
```

## 11.3.4 Developing views

In Dojo Mobile, views are containers for other user interface controls. You can find a sample template for a new view in the following location:

`<common folder>/view/TEMPLATE_NAME.html`

Views in the sample typically consist of a Heading, with Back and Action buttons, followed by the content that is usually a vertically scrollable list:

`dojox.mobile.EdgeToEdgeList`

In our sample, we customize or add the following views:

- WorkView: Used to display a list of work items.
- propertiesView: Used to display properties of the selected work item.

Dojo Mobile widgets do not inherit from `dijit/_TemplateMixin`, by default, as standard *dijits*; we will reference them using `dijit/registry` module. For this reason, we are not referencing them using the standard Dojo attach point mechanism.

### Adding WorkView

This view will contain a Back button that will navigate to MainMenu view; a list identified as `workItems` is used to display a list of work items. Add this view by following these steps:

1. Right-click the `<common folder>/view` folder, select **New** → **File**, and name it `WorkView.html`.
2. Paste the content of Example 11-3 to the newly created HTML file and save it.

*Example 11-3 WorkView HTML code*

```
<div id="WorkView" data-dojo-type="dojox.mobile.ScrollableView"
data-dojo-props="controller:'WorkView'">
    <div data-dojo-type="dojox.mobile.Heading"
data-dojo-props="fixed: 'top', back:'Back', moveTo:'MainMenu'">
        <div id="WorkView_PIID" data-dojo-type="dojox.mobile.Pane">
        </div>
</div>
<ul id="workItems" data-dojo-type="dojox.mobile.EdgeToEdgeList"
data-dojo-props='stateful:true' class="unselectable"></ul>
</div>
```

### Customizing propertiesView

We will reuse propertiesView to display attributes of the selected work item. However, it contains Edit button for which we do not have use and will hide it in our case.

1. Open `<common folder>/view/mainMenu.html` and find the `div` element with the `id=”propertiesView”` attribute.
2. Find the nested `div` element of the Edit button and add the following information to simplify hiding the button:

   ```
   id="editPropertiesButton"
   ```

   In Example 11-4 on page 427, the changed code is highlighted in **bold**.

   *Example 11-4 Add reference to Edit button*

   ```
   <div data-dojo-type="dojox.mobile.ScrollableView" id="propertiesView" >
      <div data-dojo-type="dojox.mobile.Heading" id="propertiesHeading"
   style="margin-right: 0px; margin-left: 0px; top: 0px;"
   data-dojo-props="back:'Back', moveTo:'BrowseListView'">
               <div id="editPropertiesButton"
   data-dojo-type="dojox.mobile.ToolBarButton"
   data-dojo-props='label:"Edit",defaultColor:"mblColorBlue"'
   onClick="getBrowse().editProperties();" style="float:right;">
   </div>
   ...
   ```

3. Save file.

## 11.3.5 Adding controller for WorkView

In this section, we add a controller for the WorkView created in “Adding WorkView” on page 426. It will be responsible for retrieving model objects from MILayer and modifying the view in order to display work items. We also reuse existing propertiesView to display the work item properties.

1. Right-click the **<common folder>/js/controller** folder and select **New** → **Folder** and name it `workview`**.**
2. Right-click the **workview** folder and select **New** → **File** and name it `WorkViewController.js`**.**
3. Paste skeleton of our controller, as shown in Example 11-5. We add implementation in the following steps.

*Example 11-5 Skeleton of controller*

```
define(["dojo/_base/declare","dojo/_base/lang","dojo/dom-style",
"dojo/aspect","dijit/registry", "controller/BaseController"],
function(declare, lang, domStyle, aspect, registry) {
    return declare("WorkView", [
       BaseController
   ],{
         /* Place methods here */
   });
  }
);
```

4. Add the loadWorkitems method, shown in Example 11-6 on page 428, that is responsible for performing a segue when our feature is selected in the feature list of the desktop.

*Example 11-6 Performing segue*

```
..
// /* Place methods here */
loadWorkitems:function(listItem){
       getTransitionManager().performSegue(listItem, "WorkView",
"view/WorkView.html",null,getCtrlInstance('WorkView'));
    },
..
```

5. Now, when the segue is performed, the framework will call the onAfterTransition method that is responsible for displaying the content of the view (in our case, retrieving work items and adding them to our view using the addItem method). It will also display a lock icon in case the work item is locked. Add the methods listed in Example 11-7 after the previously inserted code.

*Example 11-7 Initiation of view*

```
...
addItem: function(item,view) {
   var listItem = new dojox.mobile.ListItem({
      label: item.name,
      clickable: 'true',
      rightIcon: (item.locked) ? 'images/CheckedOut.png' : null
   });
   aspect.after(listItem,"onClick", lang.hitch(this,
"showProperties", item, listItem));
   listItem.placeAt(view.containerNode);
},
```

```
onAfterTransition:function(view, moveTo, direction, transition,
context, method){
   var destinationView = registry.byId('workItems');
   destinationView.destroyDescendants();
   commonMIlayer = getMILA();
   if (commonMIlayer.currentRepository.repositoryId !=
ecm.model.desktop.defaultRepositoryId){
commonMIlayer.setCurrentRepository(ecm.model.desktop.defaultReposito
ryId);
}
startProgressIndicator(null, false);
commonMIlayer.retrieveWorkItems(lang.hitch(this,function(wItems) {
   for (var i in wItems.items) {
      this.addItem(wItems.items[i],destinationView);
   }
   stopProgressIndicator(true);
}));
},
...
```

6. Previously, we used the aspect-oriented approach and connected a click event from a work item with the showProperties method that will use propertiesView to display attributes of the work item. Because this view is reused from other views, we programmatically change that behavior of the Back button to return to our view, hide system properties section, and the Edit button. For this purpose we add code listed in Example 11-8.

*Example 11-8 Displaying properties*

```
...
showProperties: function(wItem, listItem) {
   var sysProps = registry.byId("SystemPropertiesList");
   var propHeading = registry.byId('propertiesHeading');
   var propsView = registry.byId("propertiesView");
   var labelString = wItem.name;

   if (labelString != null &&  labelString.length > 13){
      labelString = labelString.substr(0,13) + "...";
   }
   propHeading.set('label', labelString);
   propHeading.backButton.set("moveTo", "WorkView");
   propHeading.backButton.onClick = function() {
   domStyle.set(registry.byId("SystemPropertiesList").domNode,
'visibility','visible');
```

```
   domStyle.set(registry.byId('editPropertiesButton').domNode,
'visibility', 'visible');
};

domStyle.set(registry.byId('editPropertiesButton').domNode,'visibili
ty', 'hidden');
    domStyle.set(registry.byId('systemPropertiesTitle').domNode,
'visibility','hidden');
   domStyle.set(sysProps.domNode, 'visibility','hidden');

   sysProps.destroyDescendants();
   registry.byId("PropertiesList").destroyDescendants();

   for(var key in wItem.attributes) {
      if(wItem.attributes[key]) {
         addPropertyListItem([key,wItem.attributes[key]],'user');
      }
   }
   /*scroll to top*/
   dojo.setStyle(propsView.containerNode, {
      webkitTransform: '',
      top: 0,
      left: 0
   });

   listItem.transitionTo('propertiesView');
}
...
```

**Note:** If you are not appending methods in the order they are listed, be sure the JSON syntax is not broken. Unlike other methods, the last one must not be terminated by a comma.

## 11.3.6 Integrating to sample

After developing our feature, we integrate it to the rest of the application. This integration consists of the following tasks:

- Registering the WorkView Controller
- Registering the handler for the new mobile feature
- Adding the Work feature icon

Complete the following steps:

1. Open the <common folder>/js/controller/menu/MenuController.js file.
2. Locate the constructor of MenuController. Paste the content of Example 11-9 at the end of the method referenced in the previous section.

*Example 11-9 Registration of WorkView controller*

```
constructor:function(){
   setBrowse(browse);
   getControllersManager().register('browse', browse);
   /*loading controllers*/
   dojo.require('controller/favoritesview/FavoritesViewController');
   dojo.require('controller/searchview/SearchViewController');
dojo.require('controller/searchtemplateview/SearchTemplateViewContro
ller');
   dojo.require('controller/workview/WorkViewController');
},
```

3. Locate the onAfterTransition method, find the following row, and place the registration of the Work feature before it, as shown in Example 11-10:

```
if (labelText == 'Browse'){
```

*Example 11-10 Register Work feature to the desktop list*

```
...
if (labelText == 'Work') {
   knownMenu = true;
   onClickFunction = function() {
      getCtrlInstance('WorkView').loadWorkitems(this);
   };
}
if (labelText == 'Browse'){
...
```

4. Create an icon (32x32 pixels) for the Work feature, such as in Figure 11-6; give the name **Work.png** to the icon.

*Figure 11-6 Work feature*

5. Place this file in the following folder:

```
mobileHybrid\apps\mobileHybridApp\common\images
```

Now the customization is complete and you can test and deploy it. See 11.3.7, “Packaging and deployment” on page 432

## 11.3.7 Packaging and deployment

To test your application with the simulator, see the documentation referenced in “Preparing the development environment” on page 424.

When your application works as you expect in the simulator you can build the sample as a hybrid iPhone application. We demonstrate the other option, how to build the sample as an IBM Content Navigator plug-in that is not deployed into a mobile device.

Figure 11-7 on page 433 illustrates the deployment architecture of a sample build as an IBM Content Navigator plug-in. As you can see, all of the sample code is stored in SampleMobilePlugin deployed to IBM Content Navigator. The plug-in contains sample MobileLayout that must be assigned to the desktop. When the client opens the desktop from a device browser, IBM Content Navigator starts loading the layout and checking the user-agent header sent by the browser. In the case of an iPhone device, the layout will redirect the browser to the initial web page of the sample. This page is accessed by a *plug-in resources* action. As you can see, this type of deployment still allows you to work with midtier services. You still have full potential of the web application, however, your code cannot use Worklight APIs to, for example, access a native device. Also, you must prevent same-origin-policy violations. However, you do not need to handle this when accessing midtier services because the sample detects that it runs as a plug-in and will automatically use the desktop you are accessing and prompt for login. Make sure your `<common>/js/controller/desktop/common.js` file has the enablePluginMode property set to `true`.

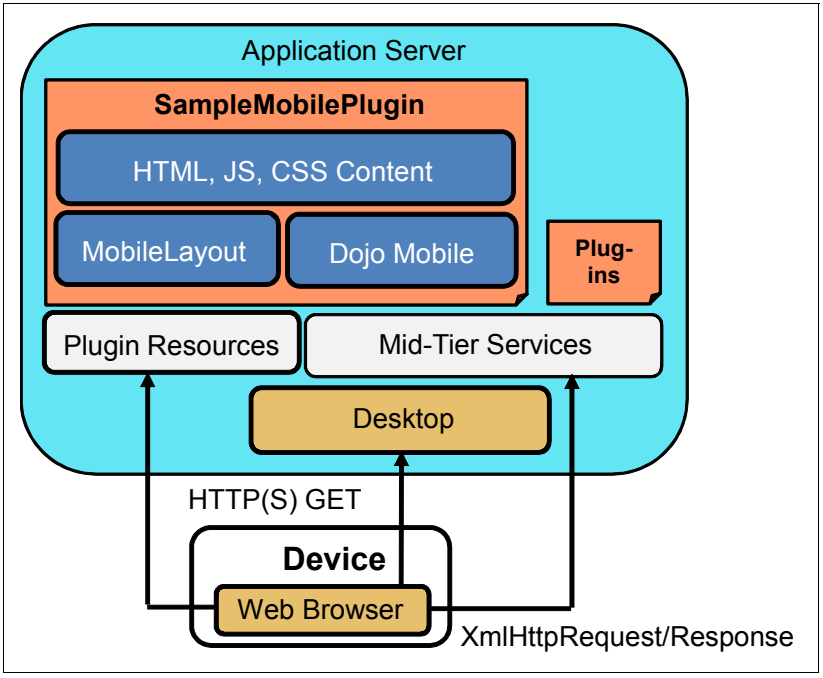



*Figure 11-7 Mobile client packaged as a plug-in*

To build sample as a plug-in, complete the following steps:

1. Open the Eclipse project that is prepared in “Preparing the development environment” on page 424.
2. Right-click the **bin** folder located directly under the `mobileHybrid` folder and select **Properties**. Open this folder from your file browser outside of Eclipse. You can find the path in **Resource** → **Location**.
3. In the bin folder, create folder named `plugin`.
4. Locate your `SampleMobilePlugin.jar` file and rename it to `SampleMobilePlugin.zip`.
5. Extract the `com` folder from the ZIP file to the `plugin` folder. The final path is as follows:

   `mobileHybrid\bin\plugin\com.`
6. Return to Eclipse and right-click the **mobileHybrid\bin** folder again. Select **New** → **File** and name it `build.xml`. Place the Ant build file as listed in Example 11-11 into this file and then save it.

*Example 11-11 Plugin Ant build*

```
<?xml version='1.0' encoding='UTF-8'?>
<project name="samplePlugin" default="plugin" basedir=".">
  <target name="plugin" depends=""
        description="generate plugin">
      <delete file="../apps/SampleMobilePlugin.jar"/>
      <delete
dir="./plugin/com/ibm/ecm/extension/sample/WebContent/mobile"/>
      <mkdir
dir="./plugin/com/ibm/ecm/extension/sample/WebContent/mobile"/>
      <copy
todir="./plugin/com/ibm/ecm/extension/sample/WebContent/mobile">
      <fileset dir="../apps/mobileHybridApp/common"/>
      </copy>
      <copy
todir="./plugin/com/ibm/ecm/extension/sample/WebContent/mobile/dojo"
><fileset dir="../dojo/dojo"/></copy>
      <jar destfile="../build/SampleMobilePlugin.jar"
            basedir="./plugin">
          <manifest>
            <attribute name="Plugin-Class"
             value="com.ibm.ecm.extension.sample.SamplePlugin"/>
          </manifest>
    </jar>
      <copyfile src="../build/SampleMobilePlugin.jar"
dest="../apps/SampleMobilePlugin.jar" forceoverwrite="true"/>
  </target>
</project>
```

7. Right-click **build.xml** and select **Run as** → **Ant Build**.
8. You can find your new plug-in JAR file in the following location:

   `mobileHybrid\apps\SampleMobilePlugin.jar`
9. Follow the steps from the product documentation (in the “Installing and deploying the sample layout plug-in” topic) referenced in “Preparing the development environment” on page 424.

**Note:** With this type of deployment scenario, users might access the `common.js` file that can contain passwords you have used for testing in aa simulator. For security reasons, change the password to something else, the sample will not use this password when deployed as a plug-in.

## 11.4 Conclusion

This chapter describes options for mobile development and presents architecture of IBM Worklight sample project that is included with the IBM Content Navigator product. It also illustrates customization of this sample and describes deployment scenarios. This chapter provides information that is useful for further extension of the sample.

12

# Extending Profile Plug-in for Microsoft Lync Server

This chapter describes how to extend the IBM Connections Profile Plug-in to work with Microsoft Lync Server. IBM Content Navigator 2.0.2 provides an IBM Connections and Profile plug-in. The plug-in works with IBM Connections Service and IBM Sametime® server. It provides a business card and Sametime awareness for users in IBM Content Navigator (ICN) UI. If you use IBM Content Navigator and Microsoft Lync Server, you might want to be able to see the status of other users online from within IBM Content Navigator. To do so, you can develop an IBM Content Navigator plug-in to extend IBM Content Navigator to show the online information of users.

This chapter covers the following topics:

- What is available in IBM Content Navigator 2.0.2
- Microsoft Lync Server and UCWA
- Example overview
- High-level design
- Implementing the Lync Plug-in
- Object-oriented design for Java and JavaScript
- Setup, installation, and enhancements

© Copyright IBM Corp. 2012, 2014. All rights reserved.

For ease of reference and distinction in this chapter, we use the following conventions:

- We refer to the existing IBM Connections and Profile Plug-in as the *Profile Plug-in*.
- We refer to the new plug-in for Microsoft Lync Server as the *Lync Plug-in*.

## 12.1 What is available in IBM Content Navigator 2.0.2

IBM Content Navigator 2.0.2 provides a sample plug-in for IBM Connections server and for IBM Sametime awareness. Typically, it has the following installation path:

```
C:\IBM\ECMClient\plugins\ProfilePlugin.jar
```

The Profile plug-in works with a sample IBM Connections service and provides a business card feature, so that when you use a mouse to hoover over a user name in the Browse view, that user's business card displays the user's information from the IBM Connections service. An example of this feature is in Figure 12-1.

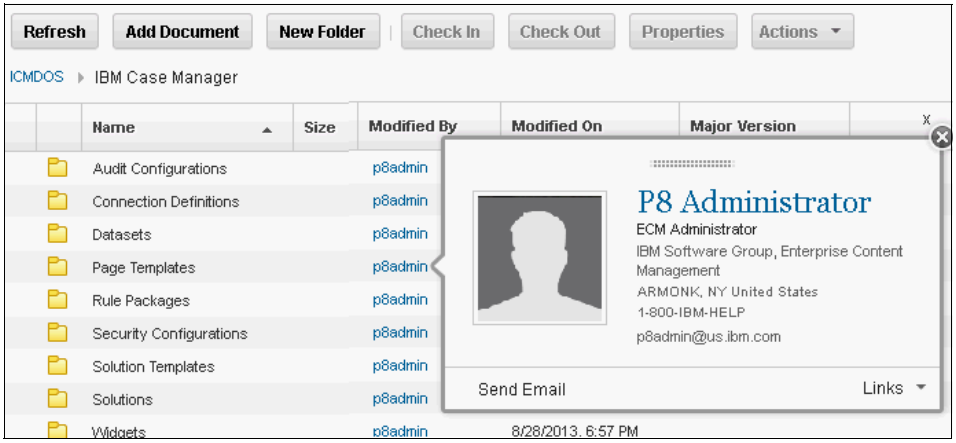



*Figure 12-1 Existing business card functionality in IBM Content Navigator 2.0.2*

The IBM Connections Profile plug-in also works with IBM Sametime server, so that a user's online status can be displayed along with the user name, and that it allows you to initiate a Sametime conversation with that user.

The Sametime functionality has been extended and supported in IBM Case Manager, a product that leverages IBM Content Navigator as the user interface platform.

Figure 12-2 and Figure 12-3 show the Document and History tabs for a sample Case Manager user interface.

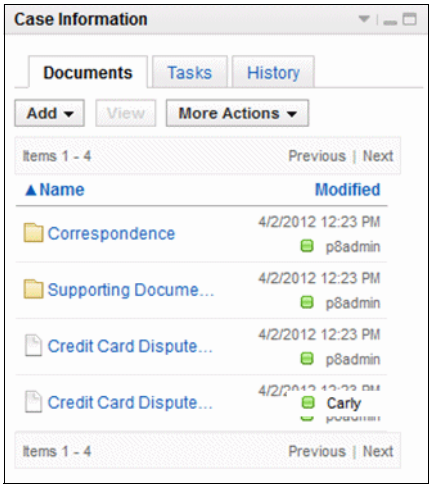



*Figure 12-2 Sample IBM Case Manager user interface: Document tab*

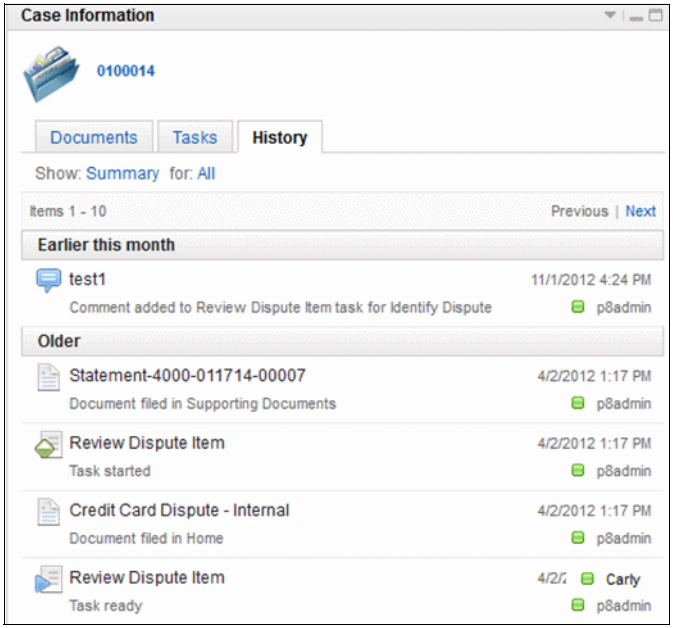



*Figure 12-3 Sample IBM Case Manager user interface: History tab*

## 12.2 Microsoft Lync Server and UCWA

Similar to the IBM Sametime server, Microsoft also provides an online communications platform named Microsoft Lync.

Microsoft Lync Server is enterprise real-time communications server software, providing the infrastructure for enterprise instant messaging, presence, voice over IP, audio, video, web conferencing, and PSTN connectivity through a third-party gateway or SIP trunk. These features are available within an organization, between organizations, and with external users on the public Internet, or smartphones.

*Microsoft Lync* is the instant messaging client used with *Microsoft Lync Server*. It is enterprise software that is targeted to corporate environments.

Microsoft Lync Server is equivalent to IBM Connections server with IBM Sametime server. Microsoft Lync Server provides the following API:

- Unified Communications Managed API (UCMA)
- Unified Communications Web API (UCWA)

### 12.2.1 UCMA

Microsoft Unified Communications Managed API 4.0 enables developers to build applications that leverage the full power of the Microsoft Lync Server 2013 platform. Applications built on UCMA 4.0 incorporate unified communications concepts such as presence, call, conversation, and conference.

UCMA is a C# API that includes development and runtime components. UCMA is used by developers to develop communication solutions for the enterprise. UCMA contains a managed code endpoint API that is based on Session Initiation Protocol (SIP). Its current version is UCMA 4.0.

### 12.2.2 UCWA

Unified Communications Web API is a Microsoft web API to communicate with Microsoft Lync Server. UCWA 1.0 is a REST API that exposes Microsoft Lync Server 2013 instant messaging (IM) and presence capabilities. UCWA 1.0 enables developers to make their enterprise applications and intranets more lively and connected to business contacts.

UCWA 1.0 is language-independent. Developers can use any programming language, for example C/C++ and Java.The API is fine-tuned for web developers

who are familiar with ordinary web technologies such as HTTP, OAuth, JSON, and JavaScript.

UCWA 1.0 is available only to customers who have Microsoft Lync Server installed.

**Note:** If you are interested in developing UCWA web applications, a good reference is to search for `UCWA` at the following location:

`http://msdn.microsoft.com/library/office/gg455051%28v=office.14%29`

## 12.3 Example overview

We build a new IBM Content Navigator plug-in with the following functions:

- Display a user's contact card and online status.
- Reflect a user's online status after it changes.

To keep things simple and easy to understand, we do not develop a production-ready, full-fledged plug-in. Rather, the new plug-in is developed with the following assumptions and limitations:

- Assumptions:
  - Plug-in works with Microsoft Lync sandbox.
  - Installation of Microsoft Lync Server is not required.
- Limitations:
  - Initiating chat conversation is not supported in the plug-in.
  - Updating of user online status is out of the scope of the plug-in.
  - Conference, phone call, video chat capabilities are not supported in the plug-in.

## 12.4 High-level design

For integration with Microsoft Lync, we use an IBM Content Navigator plug-in, In the plug-in, we build a Microsoft Lync Contact Card and Status Service.

### 12.4.1 Goals of the new plug-in

When designing and building the new plug-in, consider these design goals:

- Construct a plug-in that works with Microsoft Lync Server.
- Build the plug-in based on current profile plug-in.
- Reuse logic as much as possible.
- Avoid duplicating existing logic.
- Make the code easy to maintain.
- Avoid having to merge two profile plug-ins later.
- Make the plug-in easy to debug.

The two plug-ins, Profile Plug-in and Lync Plug-in, have similar purposes, structure and code. One plug-in deals with IBM Connections and Sametime servers. The other plug-in works with Microsoft Lync Server. They both display a business card. They both need to add decorators to fields in content list grids.

A *quick approach* is to make a copy of the existing Profile Plug-in and then overwrite it to suit the needs of the new plug-in. Although that might be a good starting point, the problem with this approach is when more features and functions are added to each plug-in, one must remember to apply the same bug fixes to the other plug-in. If we go with this approach, the code might be difficult to maintain in the future.

A good approach is to start with the quick approach to get the new plug-in to work. Then, try to merge the two plug-ins together, so we eliminate duplicate code.

A *better approach* is to start by designing with reuse in mind. We accommodate two plug-ins in the same project. We will reuse the existing code, and then extend it to work for Microsoft Lync Server.

Explaining all the details of the implementation in the chapter is not possible. So only the major concepts and areas are highlighted in this chapter.

The exact interrelationships of the two plug-ins can be better understood from the included compressed project source files. See Appendix D, “Additional material” on page 535 to learn about downloading the additional materials that are associated with this book.

The Lync Plug-in uses the response filter (introduced in Chapter 1, “Extension points and customization options” on page 3), to filter the service request to repository source. It then introduces custom formatters for particular fields in the response. IBM Content Navigator will call this response filter for service requests to repository.

The Lync Plug-in service is used to add a new server-side action to IBM Content Navigator.

### 12.4.2 Contact card or business card function

A contact card contains the following information:

- Presence
- Location
- Note

UCWA 1.0 enables a user to both publish and view information on presence, location, and note. In the current release, the API supports the standard set of presence states, such as Online, Busy, and Away. Custom presence can be viewed only through this API. Locations are user-provided strings that can be set or displayed for sharing with other contacts. For note, the API supports publishing the personal note and viewing either the personal or out-of-office note. The note information is driven by the server-side logic and the user’s calendar.

These three pieces of information are viewable for all contacts in the API.

### 12.4.3 Microsoft Lync Contact Card and Status Service

The Microsoft Lync Contact Card and Status Service will be built inside the Lync Plug-in. The service will be composed of a plug-in service, which will communicate with the Microsoft Lync Server through UCWA.

Figure 12-4 on page 444 illustrates the relationship of IBM Content Navigator, the plug-in, and the Microsoft Lync Server.

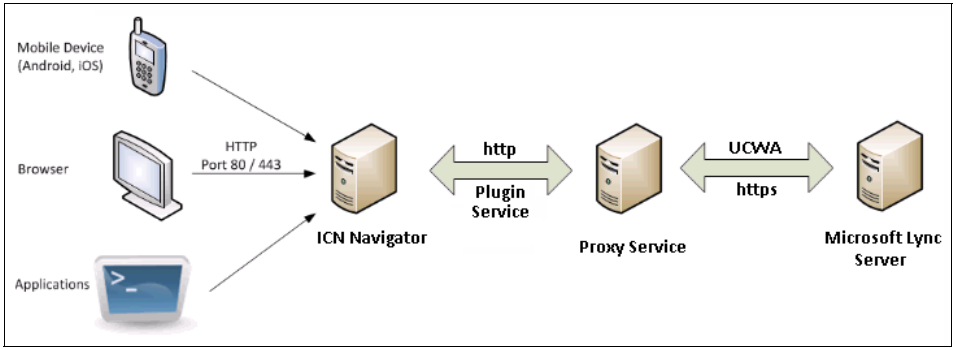



*Figure 12-4 High-level components*

# 12.5 Implementing the Lync Plug-in

In this section, we discuss the details in the implementation of the Lync Plug-in. We will look at the configuration of the plug-in, the user interface, error handling, the login process, getting the contact information, and adding the response filter.

## 12.5.1 Configuring the Lync Plug-in

The plug-in has a configuration page that accepts the following parameters:

- Microsoft Lync Service URL

  This is the URL to Microsoft Lync Service, such as the following example:

  `https://ocsrp.gotuc.net/ucwa/oauth/v1/applications`

- OAuth Token from Microsoft Lync Service

  This is the token from the Microsoft Lync service. One can obtain the OAuth token from a Microsoft Lync sandbox, after logging in to the sandbox.

- Check box for whether to show the display name (optional).

  This optional parameter is reused from the Profile Plug-in. When selected, it shows the display name instead of the short name, in places where user names are shown.

Figure 12-5 shows the configuration of the Lync Plug-in.

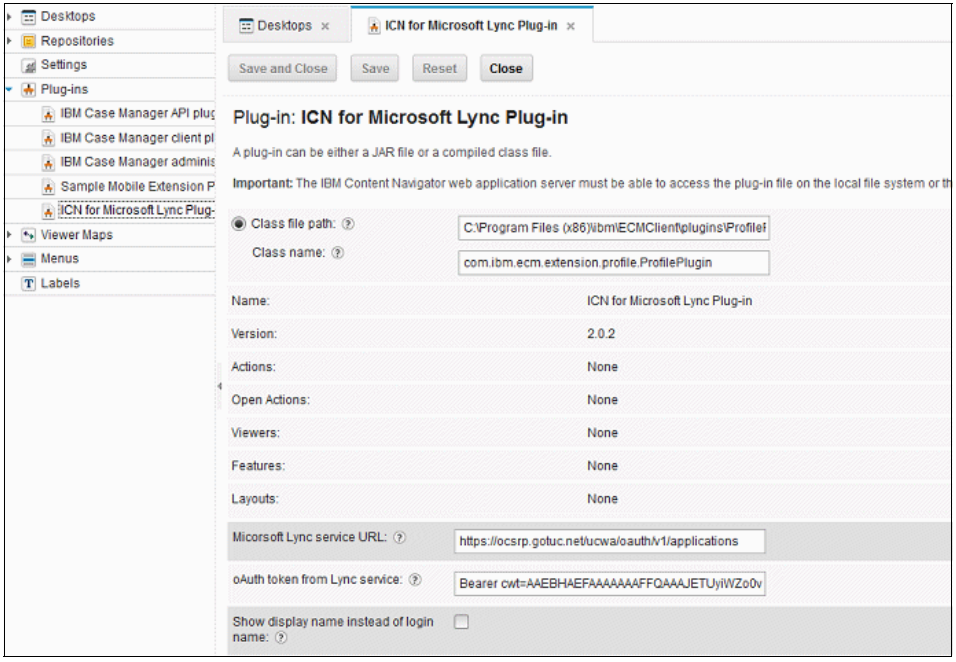



*Figure 12-5 Configuration for the IBM Content Navigator Lync Plug-in*

## 12.5.2 Microsoft Lync Service user interface

When the mouse is hovered over a field in the ListView, a business card will be displayed next to the name showing the user's contact and online information.

The field is a data field in a grid in the ListView. It can be either the detail view or the magazine view.

An IBM Content Navigator decorator is added to user name fields. The decorator is JavaScript code that displays the business card when during hover. The following fields have been added to a decorator to have extra JavaScript functionality:

- Created by
- Last Modifier

Figure 12-6 shows the user interface when the mouse is over a user's name.

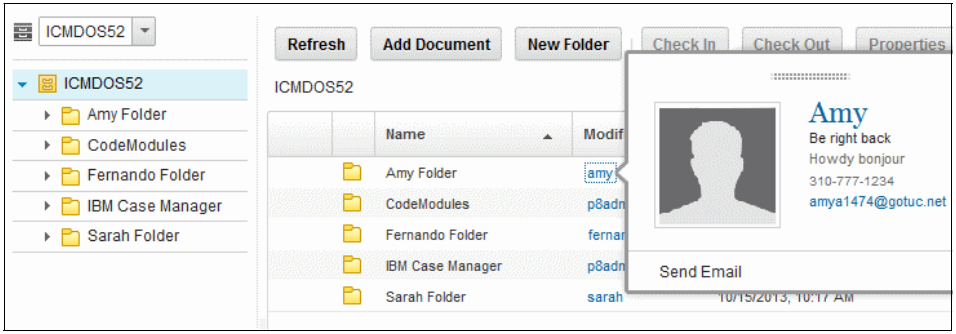



*Figure 12-6 Business Card feature with Microsoft Lync*

## 12.5.3 Error handling

The OAuth token obtained is usually valid only for a certain period of time, such as 8 hours. It expired afterward. The plug-in is designed to show a message when an error situation occurs. The contact card continues to be displayed, but an error message is also displayed, with an error code if available, and advice to the user, for example to check OAuth token or the Microsoft Lync Service URL. Figure 12-7 shows a sample error message when an OAuth token is expired.

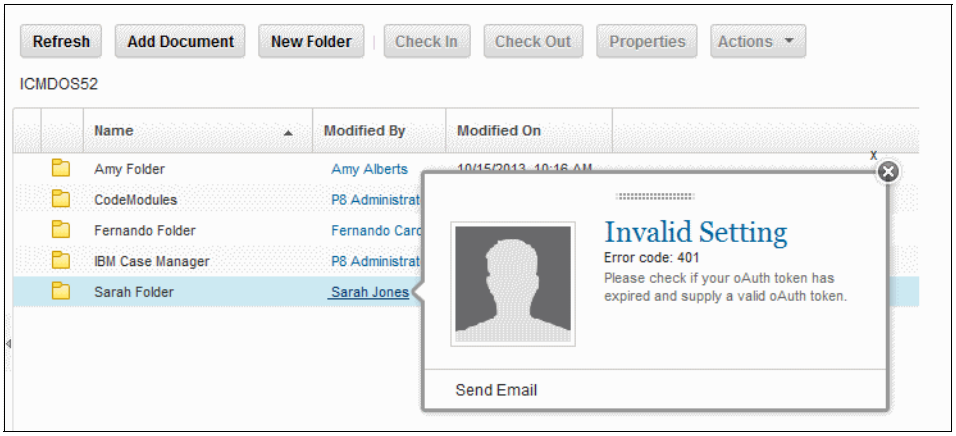



*Figure 12-7 Error handling when token expires*

## 12.5.4 Log in to Microsoft Lync Service

The Lync Plug-in takes a user's email address as input, contacts the Microsoft Lync Server, and returns the user's contact and online information as output. The first step it does is to log in to the Microsoft Lync Server.

The plug-in service maintains a LyncSession object. With every email lookup request, the plug-in service first determines if the internal Microsoft Lync session exists. If it does not exist, the plug-in attempts to log in to Microsoft Lync Server.

Login requires submission of the following information:

- Culture: This is typically the browser locale.
- Endpoint: This is a GUID that identifies the user.

If the OAuth token is missing or invalid, the Microsoft Lync Server responds with an HTTP 401 response. When the OAuth token is valid, the server responds with the basic information of the user. Example 12-1 shows the server response when the OAuth token is valid and login is successful.

*Example 12-1 Microsoft Lync Server response after a successful login*

```
{
  "culture":"en-US",   "userAgent":"UCWA Samples",
  "_links":{
    "self":{"href":"/ucwa/oauth/v1/applications/105"},
"policies":{"href":"/ucwa/oauth/v1/applications/105/policies"},
    "batch":{"href":"/ucwa/oauth/v1/applications/105/batch"},
"events":{"href":"/ucwa/oauth/v1/applications/105/events?ack=1"}
  },
  "_embedded":{
    "me":{
      "name":"Lene Aaling",
      "uri":"sip:lenea@contoso.com",
      "_links":{
        "self":{"href":"/ucwa/oauth/v1/applications/105/me"},

"makeMeAvailable":{"href":"/ucwa/oauth/v1/applications/105/me/makeMeAvailable"}
,
        "phones":{"href":"/ucwa/oauth/v1/applications/105/me/phones"},

"photo":{"href":"/ucwa/oauth/v1/applications/105/photos/lenea@contoso.com"}
      },
      "rel":"me"
    },
    "people":{
      "_links":{
        "self":{"href":"/ucwa/oauth/v1/applications/105/people"},
```

```
"presenceSubscriptions":{"href":"/ucwa/oauth/v1/applications/105/people/presenc
eSubscriptions"},

"subscribedContacts":{"href":"/ucwa/oauth/v1/applications/105/people/subscribed
Contacts"},

"presenceSubscriptionMemberships":{"href":"/ucwa/oauth/v1/applications/105/peop
le/presenceSubscriptionMemberships"},
        ...
      },
      "rel":"people"
    },
...
}
```

## 12.5.5 Getting contact information from Microsoft Lync Server

There are two ways to get a user's information from Microsoft Lync Server:

- Use the search API.
- Use the contact API.

The search API returns user information pertinent to the query string. It may return multiple results. For example, if the query is for `amya1`, several contacts might be returned, including `amya1@gotuc.net`, or `amya100@gotus.net`.

The contact API requires a user email and returns only one result for a user's contact information. The contact mechanism requires a complete email address and is the preferred way of looking up user information to reduce ambiguity.

Example 12-2 shows a response from the Microsoft Lync Server when user information is requested using the search API from the Microsoft Lync Server.

*Example 12-2 Retrieving user information based on email address*

```
https://ocsrp.gotuc.net/ucwa/oauth/v1/applications/102902378342/search?query=am
ya1@gotuc.net
{
    "moreResultsAvailable": false,
    "_links": {        "self": {            "href":
"/ucwa/oauth/v1/applications/102902378342/people/search?query=amya1@gotuc.net\u
0026limit=50"       }    },
    "_embedded": {        "contact": [{
            "uri": "sip:amya1@gotuc.net",            "sourceNetwork":
"SameEnterprise",
            "department": "Engineering",
            "title": "Software Development Engineer",
```

```
            "emailAddresses": ["amya1@gotuc.net"],
            "workPhoneNumber": "tel:+1001",
            "type": "User",            "name": "Amy E Alberts",
            "_links": {
                "self": {                    "href":
"/ucwa/oauth/v1/applications/102902378342/people/amya1@gotuc.net"
},
                "contactPhoto": {                    "href":
"/ucwa/oauth/v1/applications/102902378342/photos/amya1@gotuc.net"
},
                "contactPresence": {                    "href":
"/ucwa/oauth/v1/applications/102902378342/people/amya1@gotuc.net/presence"
},
                "contactLocation": {                    "href":
"/ucwa/oauth/v1/applications/102902378342/people/amya1@gotuc.net/location"
},
                "contactNote": {                    "href":
"/ucwa/oauth/v1/applications/102902378342/people/amya1@gotuc.net/note"
},
                "contactSupportedModalities": {                    "href":
"/ucwa/oauth/v1/applications/102902378342/people/amya1@gotuc.net/supportedMedia
"                },
                "contactPrivacyRelationship": {                    "href":
"/ucwa/oauth/v1/applications/102902378342/people/amya1@gotuc.net/privacyRelatio
nship"                }
            },
            "rel": "contact",            "etag": "78185271"
        }]
    },
    "rel": "search"
}
```


**Tip:** Currently, you might not see many of the attributes, such as phone number and department. This is not a programming error on your part. This is because the Microsoft Lync sandbox is not being properly set up for all accounts. As an example, for Amy, only `amya1@gotuc.net` has all the information. Other Amys do not have this information populated.

The information about a user must be setup on the Microsoft Lync Server so that it can be displayed. When using Microsoft sandbox server, it might not return all the information. When using your own Microsoft Lync Server, be sure to populate the fields such as workPhoneNumber.

## 12.5.6 Adding a response filter to user name fields

To display a business card over a user name, first add a response filter to certain fields in a server response. The response filter adds a decorator to certain columns in a service response. The decorators are specified in a Java class and then defined in a JavaScript class.

We set up the response filter in `ProfilePluginResponseFilters.java`. See Example 12-3. Also see Chapter 1, “Extension points and customization options” on page 3 and Chapter 7, “Implementing request and response filters and external data services” on page 233 about response filters and the best circumstances of using response filters instead of EDS service.

The business card feature is a good example of using the response filter. It adds a special JavaScript mechanism to decorate the fields returned from FileNet P8 service calls.

*Example 12-3 Specifying response filter in ProfilePluginResponseFilter.java*

```
public String[] getFilteredServices() {
  return new String[] {"/p8/search", "/cm/search", "/cmis/search",
"/p8/openFolder", "/cm/openFolder", "/cmis/openFolder"};
}

public void filter(String serverType, PluginServiceCallbacks callbacks,
HttpServletRequest request, JSONObject jsonResponse) throws Exception {
  JSONObject structure = (JSONObject) jsonResponse.get("columns");
  JSONArray cells = (JSONArray) structure.get("cells");
  if (cells.get(0) instanceof JSONArray) {
    cells = (JSONArray) cells.get(0);
  }

  int i = 0;
  for (i = 0; i < cells.size(); i++) {
    JSONObject column = (JSONObject) cells.get(i);
    String columnName = (String) column.get("field");

    if (columnName != null &&
ProfilePlugin.profileSupportedFieldList.contains(columnName)) { // to match a P8,
CM and CMIS properties
      column.put("decorator", "businessHoverCardDecorator");
      column.put("widgetsInCell", true);
      column.put("setCellValue", "businessHoverCardCellValue");
      column.put("width", "8.0em");
    }
  }
  ...
}
```

In Example 12-3 on page 450, we add a `businessHoverCardDecorator` to columns in `profileSupportedFieldList`. The decorator will be handled by the `businessHoverCardCellValue` function.

In the JavaScript module, we define the decorators for the fields. See Example 12-4. Proper garbage collection must be done when the cells are replaced on the page. This ensures that our application does not cause a memory leak for the browser.

*Example 12-4 Field decorator defined in PersonCardDecorator.js*

```
lang.setObject("businessHoverCardDecorator", function() {
  var entry = '<span data-dojo-attach-point="entry"></span>';
  return entry;
});

lang.setObject("businessHoverCardCellValue", function(gridData,
storeData, cellWidget) {
  // memory cleanup and decorator value reset when column is sorted
  cellWidget.uninitialize();
  cellWidget.entry.innerHTML = "";

  var rowId = cellWidget.cell.row.id;
  var item = this.grid.row(rowId).item();
  var valueNode = profilePlugin._getValueNode(gridData, cellWidget,
null, item);
  domConstruct.place(valueNode, cellWidget.entry);

  // Make sure we destroy hover cards when the cellWidget is destroyed
or we will leak DOM nodes
  cellWidget.uninitialize = function() {
    var objsToDestroy = cellWidget.objectsToDestroy;
    if (objsToDestroy) {
      for ( var i in objsToDestroy) {
        objsToDestroy[i].destroy();
      }
      cellWidget.objsToDestroy = null;
    }
  };
});
```

## 12.6 Object-oriented design for Java and JavaScript

For a project to succeed and for it to be easy to maintain, design it using an object-oriented principle. The design of the new plug-in involves both the middle tier in Java, and the front end in JavaScript.

### 12.6.1 Java objects of the Lync Plug-in

The Java objects of the plug-in represent the server-side extension to IBM Content Navigator. The Microsoft Lync Status Lookup Service is implemented on the Content Navigator plug-in in Java. It is designed as several Java packages to make it object-oriented. The packages are as follows:

- com.ibm.ecm.extension.lync.service package
  - LyncService: Base class that provides the service to Microsoft Lync to retrieve information
  - UserEmail: Class for email address processing
  - UserMedia: Class for fetching user's Media setting
  - UserName: Class for user name processing
  - UserNote: Class for fetching user's note information
  - UserPresence: Class for fetching user's presence information
- com.ibm.ecm.extension.lync package:
  - Constants: Defines the global constants of the system
  - LyncSession: Establishes a session to Microsoft Lync Server and creates/maintains session info
  - UserInfo: Classes for assembling user's contact and online card based on email address
  - Util: Utility class for SSL handling
- com.ibm.ecm.extension.lync.exception package
  - LyncException: Extends the Exception class and provides response code from server and an error in JSON format

### 12.6.2 Extending the existing Java classes

For object-oriented programming, we introduce a separate package com.ibm.ecm.extension.lync to group Microsoft Lync related Java classes.

Figure 12-8 shows the Java and JavaScript class structure of the project.

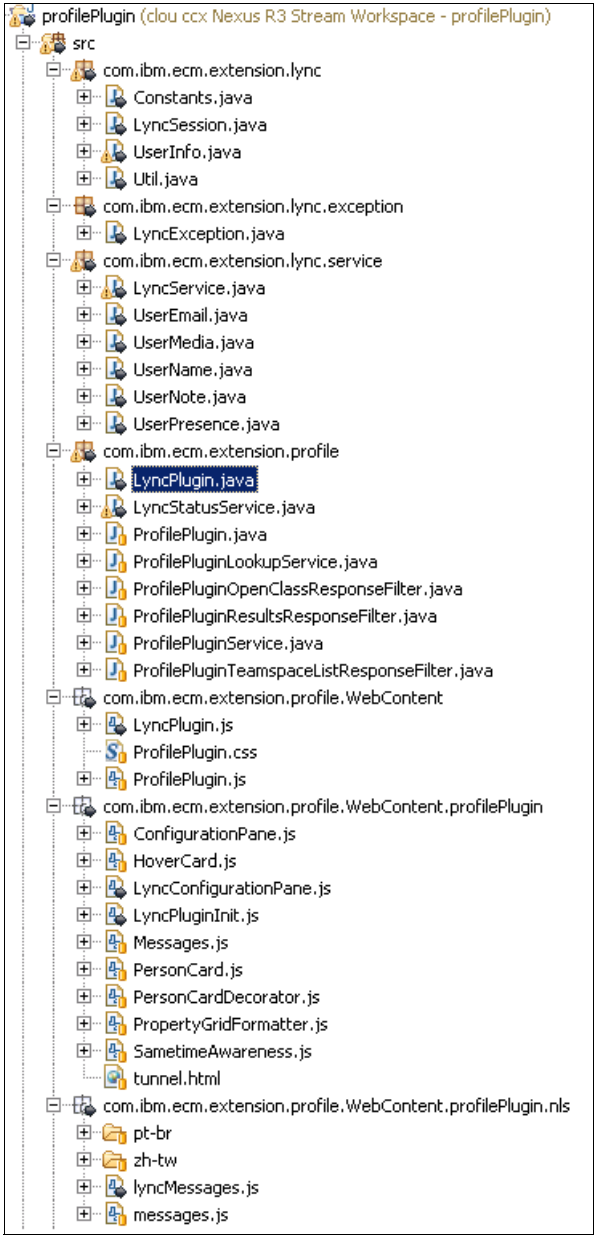



*Figure 12-8 Java and JavaScript class structure of the project*

Two Java classes are worth noting in the Lync Plug-in:

- LyncStatusService: This is the plug-in service that provides user contact card information and online information.
  - Input:
    - OAuth token
    - Application URL
    - Email address
  - Output:
    - User contact information when available and when it is set up in Microsoft Lync Server
    - Online information for note, presence, and location when it is set and provided by the user
- LyncPlugin: The plug-in definition class that defines the following items:
  - Plug-in name
  - Plug-in script: ProfilePlugin.js
  - Configuration class: profilePlugin.configurationPane
  - DojoModule: profilePlugin
  - Services and response filters

The plug-in Java class is `LyncPlugin.java`. It extends the existing `ProfilePlugin.java`. For compatibility, we use the same plug-in ID, so the code can be reused without breaking existing logic.

The Lync Plug-in defines one major service, `LyncStatusService`, which is being used to communicate with the Microsoft Lync Server. See Example 12-5.

*Example 12-5* LyncPlugin Java class extending existing Profile Plug-in

```
public class LyncPlugin extends ProfilePlugin {

  public String getName(Locale locale) {
    return "ICN for Microsoft Lync Server Plug-in";
  }

  public String getConfigurationDijitClass() {
    return "profilePlugin.LyncConfigurationPane";
  }

  public PluginService[] getServices() {
    return new PluginService[] {
      new ProfilePluginService(),
      new LyncStatusService(),
```

```
        new ProfilePluginLookupService()
      };
    }

    public String getScript() {
        return "LyncPlugin.js";
    }

    public String getVersion() {
        return "2.0.2.1";
    }
}
```

## 12.6.3 Extending JavaScript classes

Although `LyncStatusService` adds a server-side extension, we must also add an extension to the client side. There are established ways of doing object-oriented programming in Java, such as inheritance, polymorphism, and object encapsulation. In JavaScript, object-oriented programming has been difficult, because JavaScript is not strict in type. However object-oriented programming can also be designed into JavaScript. The principle of inheritance and object encapsulation can and should be applied to JavaScript also.

JavaScript classes provide the configuration and presentation of the Lync Plug-in:

- ConfigurationPane.js: Provides the configuration options for the plug-in.
- HoverCard.js: Provides pop-up information that displays when users hover the mouse pointer over an help indicator. It implements TooltipDialog class.
- Messages.js: Represents the localizable messages of the plug-in application.
- PersonCard.js: Provides the popup card when users hover the mouse pointer over a user name. This class invokes the plug-in's lyncStatusService to retrieve and render contact card information. It implements IDX PersonCard class.
- PersonCardDecorator.js: Defines the decorator classes for the detail view and magazine view in ListView. It creates the business card for IBM Connections or Microsoft Lync, or Sametime awareness based on configuration settings.
- SametimeAwareness.js: This is the JavaScript class for establishing and interacting of IBM Sametime features. It does not apply to the Lync Plug-in.

## 12.6.4 Extending the configuration pane

The new Lync Plug-in has a different set of parameters from the Profile Plug-in. For configuration, Lync Plug-in will use its own configuration pane and template, which are `LyncConfiguration.js` and `LyncConfigurationPane.html`.

In Lync Plug-in definition, we define the Dojo module class as `profilePlugin.LyncConfigurationPane`.

One early design was to have the LyncConfigurationPane extend the Profile Plug-in's ConfigurationPane. Then, a radio button was used to switch the mode between IBM Connections settings and Microsoft Lync settings. However, this introduces dependency of each plug-in.

For a clean separation of user interface and module independence, we choose to use Microsoft Lync's own template and configuration pane for Microsoft Lync.

## 12.6.5 Notes for programming with Microsoft Lync UCWA Service

Several challenges exist when programming with Microsoft Lync UCWA Service:

- Invalid security certificate in Web Service when using sandbox

  The sandbox uses an SSL certificate that is signed by an unauthorized authority. When using a real Microsoft Lync Server, this should not happen. Thus no workaround is needed.

  > **Tip:** This exception might still occur if the SSL certificate in your development, test, or production environment is self-signed.

- Changing application endpoint

  For example, in the following URL, `123456` always changes after the OAuth token expires in eight hours:

  `https://server/v1/application/123456`

- Changing user endpoint

  In the following example, `2345` changes after a certain amount of time:

  `Sarahj2345@gotuc.net`

- Authorization required even for getting user's photo

## 12.6.6 Displaying business card

The Profile Plug-in uses JSONP to retrieve user profile information. The Lync Plug-in uses a different approach. One possibility is to do it in a similar way and have the Lync Plug-in call a JSP service or external service to do email lookup. A better way is to construct a service on the server side and make the service a plug-in service.

To *not* disrupt existing logic in the Profile Plug-in, we extend the Profile Plug-in, but overwrite the `_setQueryString` method in `PersonCard` so that instead of using a JSONP service, we use the plug-in service in the Lync Plug-in. See Example 12-6.

*Example 12-6 Overwriting _setQueryString method in Lync Plug-in in LyncPluginInit.js*

```
profilePlugin.PersonCard.prototype._setQueryAttr = function(query) {
    console.log("LyncPlugin PersonCard _setQueryAttr");
    this.containerNode.innerHTML = ""; // clear content
    this._load = (this._load || dojo_lang.hitch(this,
this._setValueAttr));
    ecm.model.Request.invokePluginService("ProfilePlugin",
"lyncStatusService", {
        backgroundRequest: true,
        requestParams: {
            appUrl: profilePlugin.configuration.lync_server,
            oAuth: profilePlugin.configuration.oAuth,
            email: query.email
        },
        requestCompleteCallback: dojo_lang.hitch(this,
function(response) { // success
            console.log("request success:");
            this._load(response);
        }),
       requestFailedCallback: dojo_lang.hitch(this, function(response)
{ // failed
            console.log("lync request failed: " + response);
            this._load(response);
        })
    });
}
```

To reuse logic, we load `profilePlugin.js` from `LyncPlugin.js`, and overwrite methods of the Profile Plug-in with methods of Lync Plug-in. See Example 12-7.

*Example 12-7 Loading ProfilePlugin.js from LyncPlugin.js*

```
require([
  "dojo/_base/xhr",
  "profilePlugin/LyncPluginInit"
  ],
  function(xhr) {
    var response = {id: "ProfilePlugin", script: "profilePlugin.js"};
   var scriptUrl = ecm.model.Request.getPluginResourceUrl(response.id,
response.script);
    try {
      var scriptText;
      xhr.get({
        url: scriptUrl,
        sync: true,
        load: function(text) {
          scriptText = text;
        }
      });
      eval(scriptText);
    } catch (e) {
      console.log("_desktopLoaded", "Error evaluating JavaScript for
plugin " + response.id, e.message);

ecm.model.desktop.addMessage(ecm.model.Message.createErrorMessage("plug
in_error", [
        response.id
      ]));
    }
  }
);
```

**Note:** An alternate approach might be to relocate `profilePlugin.js` to under the JavaScript module directory so that it can be shared by both plug-ins. The advantage is that `profilePlugin.js` in the JavaScript module can be listed in the Scripts tab of Firebug, and still be possible to set breakpoints.

## 12.7 Setup, installation, and enhancements

In this section, we will focus on the setup and installation of the Lync Plug-in, and discuss future enhancements.

### 12.7.1 Considerations for development environment

Setting up a development environment can be a challenge if you are new to Microsoft Lync Server and UCWA. Some of the documentation might not be current. For Microsoft UCWA documentation, see the following web pages:

- http://ucwa.lync.com
- http://ucwa.lync.com/documentation/Resources-contact

Sandbox might not be fully configured with the necessary attributes. Your test account in the sandbox most likely might not contain information about these:

- Title
- company
- Office
- workPhoneNumber

### 12.7.2 Performance

UCWA normally requires separate calls to the Microsoft Lync Server to get individual information. Individual attributes require separate calls:

- “/note”
- “/location”
- “/presence”
- “/location” (Usually returns *empty* in the sandbox.)

**Note:** UCWA does offer a batch command, which combines several requests together in one batch. It requires parsing of boundaries, but can improve performance significantly.

### 12.7.3 Debugging the plug-in application

To debug, several techniques can be used during plug-in development:

- Set a break point in Firebug for corresponding JavaScript class.
- Check the Net tab of Firebug to examine the JSON response from the plug-in service.

- Use a Firebug plug-in such as Poster to experiment and observe the information response from the Microsoft Lync Server.
- Use Logger.logDebug (this, methodName, request, message) statements in Java class to record debug information in web server log.

## 12.7.4 Installing and setting up the Lync Plug-in

To install and set up the Lync Plug-in, complete the following steps:

1. Extract the plug-in to `C:\IBM\ECMClient\plugins` and create the following accounts on your Microsoft LDAP directory:
   - Amy
   - Fernando
   - Sarah
2. Log in to IBM Content Navigator as each individual user in step 1, and create a folder with each account.
3. Be sure your IBM Content Navigator server can access the Microsoft Lync sandbox. Otherwise, a ConnectionTimeout exception might be issued.
4. Log in to Microsoft Lync sandbox using your Microsoft Live account:

   `https://ucwa.lync.com/login`
5. Proceed with the Interactive demo; be sure to click **Start Subscription**, enter a message, and change the online status of Amy and Fernando.
6. Click **My OAuth Tokens** and copy the first token value.
7. From the IBM Content Navigator Administration Desktop, remove the Profile Plug-in if it already exists. Create a new plug-in. Enter the following parameters:
   - Class file path: `C:\IBM\ECMClient\plugins\ProfilePlugin`
   - Class name: `com.ibm.ecm.extension.profile.ProfilePlugin`
8. Select the `Class file path` radio button next. Click **Load**. Enter the following two configuration parameters:
   - Microsoft Lync Service URL:

     `https://ocsrp.gotuc.net/ucwa/oauth/v1/applications/`
   - OAuth token: Value from Microsoft Lync sandbox
9. Click **Save** and **Close**.
10. Refresh the IBM Content Navigator page. Go to the Browse feature. Hover the mouse over the user names next to the folder names that you created previously.

When you log in to Microsoft Live website, you might see the following message:

```
Sorry! We are having issues logging in right now. Please try again
later.
```

This means the sandbox website is not functioning, probably because of too many users or sessions. Try again later to obtain the OAuth token.

If you encounter the SSL certificate exception, run the logic in the `com.ibm.ecm.extension.lync.util.trustAllSSL` method.

**Note:** The Microsoft Lync sandbox at the following location might occasionally have sign-in problems. If you cannot log in to the website, return to it for testing at a later time:

`https://ucwa.lync.com`

### 12.7.5 Future enhancements

The plug-in is intended to be a starting point for you to provide user contact and online information from Microsoft Lync Server. Future enhancements can be made to the plug-in, as in these examples:

- Improve the response time of proxy service.
- Update the user's online status and messages without refreshing the Browse view.
- Add a user status icon next to the user photo.

## 12.8 Conclusion

In this chapter, we discuss how to extend the existing Profile Plug-in in ICN 2.0.2 and adapt it to be used with Microsoft Lync Server. We examine the REST API calls to Microsoft Lync Server to obtain contact information. We review the Microsoft Lync Server sandbox that provides a test platform to the Microsoft Lync service. We extend the Profile Plug-in to work with the Microsoft Lync Server sandbox, and can display a user's online information and status in ICN grids in the Browse view when the mouse is over a user name field.

We designed the code using an object-oriented approach. We designed the project so that the same plug-in can be used for both IBM Sametime and Connections service, and also for Microsoft Lync Server. This design reduces maintenance costs and avoids the challenge of having to support both services. The object-oriented design is applied to both Java in the midtier, and also the front-end JavaScript. The plug-in can be served as the base for integrating user online awareness in ICN for the Microsoft Lync Server.

# Part 3

# Deployment, debugging, and troubleshooting

This part has deployment considerations for systematically deploying and updating a solution to the production environment. The part also covers debugging and troubleshooting topics. This part contains the following chapters:


- Chapter 13, “Component deployment” on page 465
- Chapter 14, “Debugging and troubleshooting” on page 495


© Copyright IBM Corp. 2012, 2014. All rights reserved.

13

# Component deployment

A critical piece of managing your Enterprise Content Management (ECM) environment is the ability to successfully deploy and update a solution from your development environment to a production environment. The complexities and multiple components of your IBM Content Navigator environment, such as document classes, custom menus, and desktop configuration, require specific considerations for managing your application through all levels of system deployment. In this chapter, we explore the deployment considerations of rolling out such a solution.

This chapter covers the following topics:

- Strategies for managing a production environment
- Identifying the components of your application
- Manual deployment of an IBM Content Navigator solution
- Injecting a level of predictability in your environment
- Using export and import to deploy a Content Navigator solution

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# 13.1 Strategies for managing a production environment

An IBM Enterprise Content Management (ECM) environment, and any portion of your IT application infrastructure, are likely to have multiple environments that are used to support your applications and engaged for different purposes. Your ECM environment might contain several types of repositories, mirrored images for backup and disaster recovery, and clone environments used for development, quality assurance, and testing.

For an IBM Content Navigator deployment, any application configuration or development must be properly tested and verified before it is used in a product environment. Many approaches are available to manage the development and testing process for applications before deploying to a production environment. Often, components exist that an application might depend on for normal operation.

Review the following areas for dependant components within you application before deploying or migrating your application to and from various environments.

- Database entries, tables and connections
- Application server environments
- Infrastructure or hardware dependencies
- Directory services and user/group identification
- Backup and recovery strategies

When managing your production environment, be sure to include all of your applications, your development and testing environments, and all dependent environments in your support plan. When managing your production IBM Content Navigator deployment, have a clear understanding of each of the components of the application and how the application might interact with other areas of your IT environment.

Understanding these dependencies and interactions can help support the variety of activities occurring in your IBM Content Navigator environment including development, testing, backup and recovery, and production use. It can also help to support seamless promotion of your application between the various environments within your IT infrastructure.

## 13.2 Identifying the components of your application

Promoting a solution from one environment to another should be a well orchestrated and thoroughly tested procedure. It might contain many automated steps, bundled in a script for example. It might also contain several manual steps. Regardless of these steps, understanding the application's distinct components is important. Also important is to identify how these components are represented from a persistence aspect, and also the hierarchy and dependencies of each component.

### 13.2.1 Component hierarchy

Using IBM Content Navigator components, you can create custom and targeted applications to satisfy focused requirements. Those components interact with each other so they depend on the existence of each other. Furthermore, clear dependencies must be adhered to so that the application can be managed, deployed, and function correctly.

Figure 13-1 on page 468 shows several component dependencies in the composition of an IBM Content Navigator desktop. The figure implies that configuration details, for example desktop name and description or application name, do not require additional component configuration. Deploying the desktop to an environment is simply populating these values as part of the deployment. Nevertheless, there are other components, which are referenced by this desktop, that are listed in the diagram, for example a viewer map. VIewer maps can be shared among many desktops, and changing the map changes the configuration of all desktops that reference it. Because of this dependency, be sure you understand that to deploy a desktop, you must deploy the view map that it references *before* you actually deploy the desktop. This way, when the desktop is deployed, it already finds that the viewer map is available and it uses the existing reference.

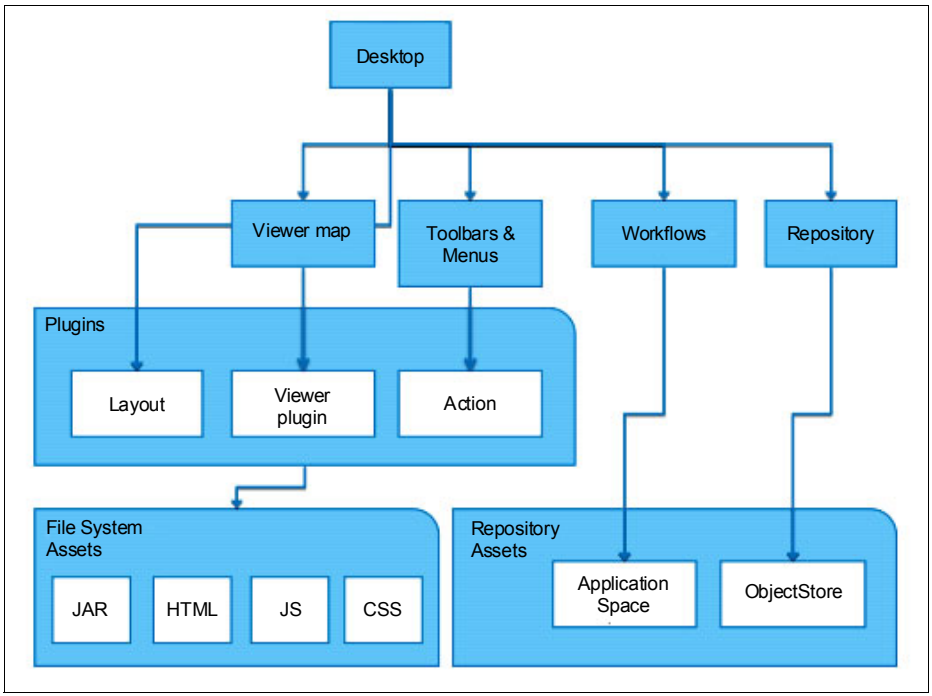



*Figure 13-1 IBM Content Navigator component dependency graph*

**Tip:** Figure 13-1 does not try to convey that to deploy a desktop, you must deploy a viewer map for example. What it does show is that, if the desktop you want to deploy has a dependency on a custom viewer map (not available by default), you *must* deploy the custom viewer map first, if it does not already exist, before deploying the desktop.

By using the same deduction mechanism, you might realize that, to formulate a deployment policy, you need to use a bottom-up approach. We start with deploying components that have zero dependencies (in Figure 13-1) these components can be JAR files, for example) and then proceed with components that have a single dependency. Using a metaphorical loop invariant, we can say that we can deploy only an *n-th* level component when all n-1 level components, which it directly depends on, have been successfully deployed.

Keep in mind that deploying a viewer plug-in, for example, can contain many steps. For example, we need to deploy the plug-in JARs and place the necessary JARs in the `WEB-INF/lib` folder. What should hold true is that, when we divide the deployment of the component into multiple steps, none of these steps should rely on deployment of another component that is not yet deployed.

## 13.2.2 Persistence characteristics

Another aspect of the deployment is the persistence layer. This mean identify where the component is stored. In IBM Content Navigator based applications, this location where the component is stored can be a custom object in an IBM FileNet Content Manager repository, a row in an IBM DB2® table, or maybe even a file on the server's file system. In summary, to develop a deployment procedure, you must know what is necessary to *lift* a component's footprint from one environment and *lay* it on another. Sometimes, automating the steps might not be possible. Therefore, an important task is to identify them and provide the necessary manual steps in your deployment procedure.

## 13.2.3 Database layer

Because of its inherent openness, IBM Content Navigator lays out a rather simplistic database schema. Therefore, administrators can more easily understand how components are stored and represented in the database. As a result, database scripts can be written to deploy any component you want, in a programmatic way. Part of the simplicity is because IBM Content Navigator has a user-friendly identifier-based model.

IBM Content Navigator uses one database table to store state information. In our case, this database is called ICN_DB and our schema is named navigator. The table contains two columns named ID (String) and ATTRIBUTES (CLOB). The ID row has the following format:

```
[COMPONENT_TYPE].navigator.[COMPONENT_ID]
```

For example, if we want to define a repository, we can use the following ID:

```
repository.navigator.MyCompanyP8
```

The first part defines this ID as a repository. The second part applies to the navigator user interface. The third part is the ID of the repository, MyCompanyP8. IDs cannot contain spaces and other characters, although exceptions to this rule do exist.

Certain IDs in the table define the default behavior of the product. For example, the following ID defines the default values when the user configures a new FileNet P8 repository.

```
repository.navigator.p8defaultsettings
```

Examining the rows of the table helps you become acquainted with the persistence layer of IBM Content Navigator.

The second column of the table is named ATTRIBUTES. This column holds the definition of the object that is defined in the corresponding ID column. The format is as follows:

```
attributeName1=attributeValue1;...;attributeNameN=attributeValueN
```

It consists of a collection of key-value pair assignments and is delimited by semi-colons. Look more closely at the attributes for the repository that we started defining previously:

```
objectStoreName=MyCompanyP8;type=p8;addAsMajorVersion=true;serverName=i
iop://server/FileNet/Engine
```

As you see, the values you enter in the user interface get flattened in this manner and saved in the database. This enables for a precise application deployment procedure.

**Important:** IBM Content Navigator depends on the integrity of the information that is stored in the database tables. Use caution when you deploy your components through a database. A mistake might break the production user interface. Always stage your deployments, and always take a backup of your database before you make any changes.

## 13.3 Manual deployment of an IBM Content Navigator solution

Completing your dependency graph can provide you with a good picture of the steps that must be orchestrated so that your deployment can be successful. We explore the deployment details from the bottom up.

The goal of this section is not to list every single component and file, row or object in the various persistence models used by IBM Content Navigator. The goal is to give you the tools to identify where your component is stored, and how to identify the necessary tasks to deploy it to a separate environment, without having to manually define it in the target environment.

### 13.3.1 Repository dependencies

We do not spend a lot of time describing repository dependencies, because they are not directly related to IBM Content Navigator. The assumption here is that whatever repository-level assets your application depends on, they were deployed before you start deploying your IBM Content Navigator components. The assets include document classes in IBM FileNet Content Manager, item

types in IBM Content Manager, folders, search and entry templates, and others. Follow the deployment guidelines of the respective repositories to ensure the success of this step.

## 13.3.2 IBM Content Navigator desktop

We devote this section on the desktop because it is the top-most component in the product hierarchy. We first cover the deployment details of a component on which desktops depend, and then we demonstrate how to deploy a desktop.

### Plug-in

Deploying a plug-in consists of two steps:

1. Deploy the built JAR file to the correct folder in the target environment. This step is described in multiple places in this book.
2. Insert the appropriate row in the database to register the plug-in programmatically, without having to use the user interface.

A plug-in is registered once and is made available to all of IBM Content Navigator. To define it, we add one row to the database table.

The following information must be present:

- ID

  `plugin.navigator.MyPlugin`

- ATTRIBUTES
  - version: The version of the plug-in from the `MANIFEST.MF` file of the JAR.
  - filename: A URI that IBM Content Navigator uses to find the JAR file. For example, it can be `file:/// or http://`.
  - name: Descriptive name of the plug-in.
  - configClass: The class from the JAR that contains the configuration user interface for the plug-in.
  - configuration: The configuration state of the plug-in. This information can be a scalar (just a single string value), or it can be a complex data structure represented in JSON.

We examine an EDS plug-in, for instance. The following example shows its corresponding row in the database:

```
"plugin.navigator.EDSSupportPlugin","version=1.0;filename=http://nexusd
emo/navigator/plugins/edsPlugin.jar;name=External Data Services
Support;configClass=edsPlugin.ConfigurationPane;configuration=http://ne
xusdemo/sampleEDSService"
```

**Tip:** Configuration parameters for a plug-in usually contain environment-specific values (for example a web service endpoint that is different for each environment). So, when we dump the row from the originating environment, we might need to replace the text values with information that is valid in the target environment before we insert and update the row.

After defining the plug-in in the target environment, we must notify the navigator application that the new plug-in exists. To do this step, we insert the ID of the plug-in (in this case EDSSupportPlugin), to the `'application.navigator'` row. This row is used by many components in the application, so we update it in the target environment.

`'application.navigator','...;plugins=...,EDSSupportPlugin;...'`

## Layout

IBM Content Navigator includes sample code that demonstrates how to develop a custom layout. A layout must be registered as a plug-in and then referenced in the desktop where it is used.

The first step is to deploy the plug-in. This is described in “Plug-in” on page 471. Follow the instructions of how to deploy the plug-in and then return here for the remaining steps.

Assuming that the plug-in JARs have been deployed and the plug-in has been registered in the database (plugin.navigator.SampleLayoutPlugin), we now need to make a reference to the layout in the desktop where we want it to appear.

The layout gets its name from the package of the class that extends PluginAction. We assume here that our class has the following name according to the sample that is provided:

`com.ibm.ecm.extension.sample.SamplePluginLayout`

We locate the row in the database that defines the desktop that we want to deploy this layout to; if this is a new desktop, we basically pull it from the originating environment. Perhaps our desktop is defined as follows:

`desktop.navigator.MyDesktop`

We get its ATTRIBUTES value and look for the key-value pair with the key equal to `layout`. We set that to the following line:

`layout=com.ibm.ecm.extension.sample.samplepluginlayout`

We update the row in the table or insert the row if this desktop is new. We recycle the navigator application in WebSphere.

## Viewer map

Deploying a viewer map is a little more complex. A viewer map consists of a collection of items, each mapping a document MIME type to a viewer that is available in IBM Content Navigator. Each desktop can have only one viewer map assigned to it at any given time.

We establish a four-step process to deploy a viewer map programmatically:

1. Deploy the mappings.

   Deploying the mappings requires one row INSERT statement for each mapping. If, for example, the ID of our viewer map is `MyViewerMap`, and we want to map AFP IBM Content Manager OnDemand documents to the `afp2pdf` viewer, we insert the following information to the database:

   ```
   "viewerMapping.navigator.MyViewerMap0","contentTypes=application/afp
   ;serverType=od;viewerName=afp2pdf;id=MyViewerMap0"
   ```

   The same is true for line data and the line data Applet Viewer:

   ```
   "viewerMapping.navigator.MyViewerMap1","contentTypes=application/lin
   e;serverType=od;viewerName=lineDataApplet;id=MyViewerMap1"
   ```

   Notice the number scheme. The ID column must adhere to this naming convention:

   ```
   viewerMapping.navigator.${ViewerMapName}[\d]
   ```

   Therefore, we concatenate the name we want to give to our viewer map with a digit, which increments each time we add a new mapping. Although the ordering of the mappings in the viewer map do matter (for precedence rules), this numbering scheme *does not* determine this ordering. This task is done in step 2.

2. Deploy the viewer map definition.

   After our viewer mappings are deployed, we define our viewer map by adding the following row to the database.

   ```
   "viewer.navigator.MyViewerMap","name=MyViewerMap;description=My
   Viewer Map;mappings=MyViewerMap1,MyViewerMap0"
   ```

   Notice how we used `MyViewerMap` to associate this viewer map with all the mappings that we deployed in the previous step (MyViewerMap0, MyViewerMap1). Also notice how the `mappings` attribute lists `MyViewerMap1` before `MyViewerMap0`. This way is how precedence is established.

3. Reference the newly deployed viewer map from the desktop that will use it.

   In this step, we assign the viewer map to a desktop. To do this step, we either update the row of the existing desktop or we insert a new row if the desktop is being deployed for the first time. The following example assumes our desktop is named `MyDesktop`:

   `"desktop.navigator.MyDesktop","...;viewer=`**`MyViewerMap;...`**`"`

4. Declare the viewer map in IBM Content Navigator so it is available to be referenced by desktops.

   As with other components, for the viewer map to be listed in the Administration Desktop when you click Viewer Maps, declare it as a global navigator component. We do that by updating the application.navigator row by adding to the viewer attribute:

   `"application.navigator","...;viewers=default,`**`MyViewerMap,...;...`**`"`

**Tip:** The viewer attribute is comma-separated in case multiple viewer maps exist.

You can still be successful if you deploy the steps out of order. However, if you do them out of order, be sure that you stop the IBM Content Navigator application in WebSphere, and do your deployments (in whatever order you want); when you complete the four steps, start the application again.

### *Reserved viewer and repository type IDs*

Previously in the example, we used certain IDs; but, how they are declared might not be immediately clear. For example, we used `serverType=od` and `viewerName=lineDataApplet`. IBM Content Navigator makes some server types and viewer IDs available by default:

- Server types:
  - `od`
  - `cm`
  - `p8`
  - `cmis`
- Viewer IDs:
  - `afp2pdf`
  - `lineDataApplet`
  - `browser`
  - `adobeReader`
  - `appletViewer`
  - `filenetViewer`
  - `iccViewer` (available only if the ICCViewer plug-in is installed)

Consult the official documentation for more information about the functionality that is provided by each viewer and repository type.

## Repository

As stated in 13.2.3, “Database layer” on page 469, each repository occupies one row in the database, and we must use the `"application.navigator"` row to make the repository available for referencing. As an example, we focus on a fictitious P8 repository named MyRepository. We define it by using the information in Example 13-1.

*Example 13-1 Information for definition*

```
"repository.navigator.MyRepository","folderNameProp=FolderName;objectSt
oreName=P8ObjectStore;type=p8;searchFilteredFolderProperties=;addAsMajo
rVersion=true;objectStore=P8ObjectStore;timeoutInSeconds=0;floatOp=;sta
tusDocDeclaredRecord=true;serverName=iiop://<ip/host>:<port>/FileNet/En
gine;searchFilteredDocumentProperties=;statusWorkItemLocked=true;folder
DefCols={NAME},ContentSize,LastModifier,DateLastModified,MajorVersionNu
mber;matchAll=true;statusDocMinorVersions=true;integerOp=;idOp=;statusW
orkItemDeadline=true;documentSystemProperties=Creator,DateCreated,LastM
odifier,DateLastModified,Id,IsReserved,IsCurrentVersion,MajorVersionNum
ber,MinorVersionNumber,ContentSize,MimeType;statusDocCheckedOut=true;pr
otocol=Navigator;annotationSecurity=inherit;docNameProp=DocumentTitle;s
tringOp=;datetimeOp=;folderSystemProperties=Creator,DateCreated,Id,Path
Name;booleanOp=;searchDefCols={NAME},ContentSize,LastModifier,DateLastM
odified,MajorVersionNumber;objectOp=;searchMaxResults=0;connectionPoint
=P8ConnPt1:1;checkinAsMajorVersion=true;name=My Repository"
```

Many configuration attributes are required for correctly defining a repository. You can configure all attributes with the IBM Content Navigator user interface in the originating environment, and then export the row and import it to the target environment.

Finally we must declare the repository so that IBM Content Navigator can make it available for other components, such as desktops and so on, to be able to reference it (which means allow it to be listed in drop-down menus for selection). We update the following row by adding the attribute (or appending to it because it will most likely exist):

`"application.navigator","...;repositories=...,`**`MyRepository`**`,...;..."`

We must recycle the navigator application in WebSphere after all our database changes are complete.

## Toolbar and menu

Toolbars and menus are similar in function and deployment also. Optionally, toolbars and menus can contain menu items stemming from plug-ins. In this case, we deploy the plug-in as described in previous chapters. After that step is complete, we follow a similar procedure as with the other components. First, we define our menu and then we declare it so that it is available to be referenced from a desktop for example. To define it, we look for a row in the database table of our originating environment that starts with `menu.navigator` and ends in the name that we assigned to it when we first set it up. In this case, we work with a menu named MyMenu. The row is similar to the following example:

```
"menu.navigator.MyMenu","items=RefreshGrid;typeLabel=Content list
toolbar;name=My Menu;type=ContentListToolbar;description=described the
menu.;id=MyMenu"
```

The only piece that might be ambiguous is how to decide on the values to populate the `items` attribute (assuming you are not taking what is there in one environment and adding it to another). This attribute is a collection of menu items and actions. To see what value you can use for an item that you can see in the user interface, hover on it and use the value that is shown as its ID. Figure 13-2 shows the corresponding menu item from the previous example.

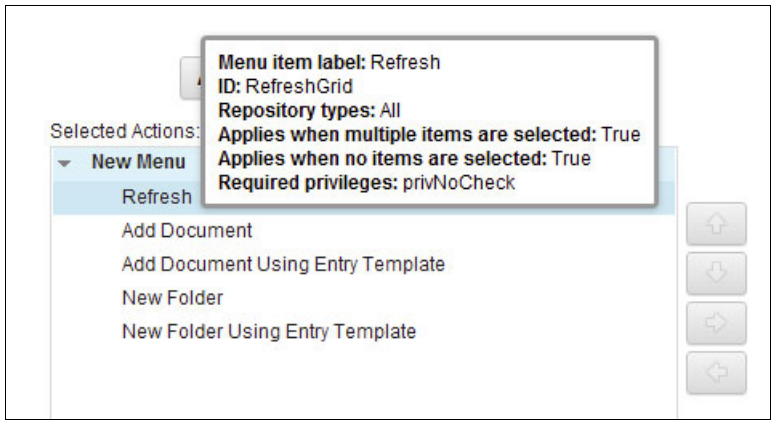



*Figure 13-2 The Refresh menu item corresponds to the ID RefreshGrid*

Often, after we define our component, in this case the menu, we need to declare it with the application to become available for referencing. We do this step in the `application.navigator` row:

```
"application.navigator","...;menus=...,MyMenu,...;..."
```

A final step is optional, depending on whether we need to use the deployed menu immediately or not. A menu is a component that can be assigned to a desktop to

further customize its user interface. So far, we defined and declared the menu but did not assign it to a desktop. We do that by assigning it to one of the menus that is available in the desktop (for example ContentListToolbar). See “Desktop” on page 477 for more details.

## Desktop

Finally, after all the dependencies are resolved, you can deploy the desktop. To deploy, one row must be updated and another must be inserted. As usual, first we insert the row which contains the definition of the desktop. Our SQL statement is shown in Example 13-2.

*Example 13-2 SQL statement*

```
INSERT INTO navigator.CONFIGURATION VALUES
('desktop.navigator.MyCompanyDesktop',
'SystemItemContextMenu=DefaultSystemItemContextMenu;fileIntoFolder=fals
e;BrowseToolbar=DefaultBrowseToolbar;ContentListToolbar=MobileDesktopco
ntentlisttoolbar;SearchTemplateContextMenu=DefaultSearchTemplateContext
Menu;loginLogoUrl=;showGlobalToolbar=false;FavoritesToolbar=DefaultFavo
ritesToolbar;culturalCollation=false;passwordRulesUrl=;applicationName=
;appSpaceLabels=;actionHandler=ecm.widget.layout.CommonActionsHandler;f
eatures=browsePane,searchPane;TrackerQueueContextMenu=DefaultTrackerQue
ueContextMenu;SelectObjectFolderContextMenu=DefaultSelectObjectFolderCo
ntextMenu;ItemContextMenu=MobileDesktopdocumentcontextmenu;repositories
=MyCompanyRepository;name=MyCompany
Desktop;description=;AddFolderAttachmentContextMenu=DefaultAddFolderAtt
achmentContextMenu;ProcessQueueContextMenu=DefaultProcessQueueContextMe
nu;TeamspaceContextMenu=DefaultTeamspaceContextMenu;showThumbnails=true
;SelectObjectToolbar=DefaultSelectObjectToolbar;bannerBackgroundColor=;
TeamspaceTemplateContextMenu=DefaultTeamspaceTemplateContextMenu;banner
ApplicationNameColor=;WorkItemDocumentContextMenu=DefaultWorkItemDocume
ntContextMenu;bannerLogoUrl=;showSecurity=false;FavoriteFolderContextMe
nu=DefaultFavoriteFolderContextMenu;bannerMenuColor=#FFFFFF;AttachmentI
temContextMenu=DefaultAttachmentItemContextMenu;FolderContextMenu=Defau
ltFolderContextMenu;TeamspaceFolderContextMenu=DefaultTeamspaceFolderCo
ntextMenu;layout=ecm.widget.layout.NavigatorMainLayout;TeamspacesListTo
olbar=DefaultTeamspacesListToolbar;MixItemsContextMenu=DefaultMixItemsC
ontextMenu;AttachmentToolbar=DefaultAttachmentToolbar;FavoriteTeamspace
ContextMenu=DefaultFavoriteTeamspaceContextMenu;InbasketToolbar=Default
InbasketToolbar;otherFeaturesDefaultRepository=MyCompanyRepository;view
er=MyViewerMap;ObjectStoreFolderContextMenu=DefaultObjectStoreFolderCon
textMenu;WorkItemFolderContextMenu=DefaultWorkItemFolderContextMenu;Fav
oriteSearchTemplateContextMenu=DefaultFavoriteSearchTemplateContextMenu
;workflowNotification=false;BannerToolsContextMenu=DefaultBannerToolsCo
ntextMenu;TemplatesListToolbar=DefaultTemplatesListToolbar;GlobalToolba
```

```
r=DefaultGlobalToolbar;AttachmentFolderContextMenu=DefaultAttachmentFol
derContextMenu;loginInformationUrl=;SearchContextMenu=DefaultSearchCont
extMenu;defaultRepository=MyRepository;BannerUserSessionContextMenu=Def
aultBannerUserSessionContextMenu;SelectObjectItemContextMenu=DefaultSel
ectObjectItemContextMenu;disableAutocomplete=false;isDefault=No;AddDocu
mentAttachmentContextMenu=DefaultAddDocumentAttachmentContextMenu;Versi
onsContextMenu=DefaultVersionsContextMenu;FavoriteItemContextMenu=Defau
ltFavoriteItemContextMenu;FavoritesContextMenu=DefaultFavoritesContextM
enu;UserQueueContextMenu=DefaultUserQueueContextMenu;messageSearchUrl=;
defaultFeature=browsePane;TeamspaceToolbar=DefaultTeamspaceToolbar;Inba
sketToolbarP8=DefaultInbasketToolbarP8');
```

**Attention:** This example stems from a complex desktop configuration that has many custom components. *Do not* insert it into your database without replacing many of the components with the components that you have already deployed.

We must also be sure to do the next step of this process, which is to declare the new desktop so that navigator makes it available to users in the user interface. We modify the `application.navigator` ID in the table by adding the ID of our new desktop (in this case MyCompanyDesktop) to the following key-value pair:

```
...desktops=admin,Demo,MyCompanyDesktop;...
```

We restart the navigator application in WebSphere and look for the new desktop.

**Tip:** The ATTRIBUTES column can be overwhelming to read because it becomes verbose. To fully understand it, you can export the whole table to a delimited file and open it with an appropriate viewer. If you instruct the viewer to delimit on commas and semicolons, the viewer shows you each of the key-value pairs that make up each component in IBM Content Navigator.

## 13.4 Injecting a level of predictability in your environment

As shown in the previous sections, developing a deployment strategy to promote an application that is designed on IBM Content Navigator can be complex. The complexity arises from having many components, which have been designed to address a wide variety of requirements and use cases. Nevertheless, creating such a deployment strategy and procedure is always rewarding, because

between major and minor releases, the application needs to be redeployed many times. The procedure can save your company many hours of work.

Be aware that we are working with many pieces of software. Many pitfalls exist that we must describe here; not all of them are related to IBM Content Navigator specifically. As much as possible, we need to bring a level of uniformity to our environments to be able to troubleshoot and fix issues quickly. Several layers that you can begin standardizing are as follows:

- File system
- Application server
- Database
- Content repository

## File system

Although much planning is required, arranging key folder paths to match one environment to another environment can provide a big advantage in the future. You might want to standardize on the following directories:

- WebSphere Application Server installation directory
- JRE directory
- JDK directory
- Library directories that are operating-system-specific (such as `/usr/lib` and `c:\windows\system32`)
- Custom application directory (for example `/opt/MyApp`)

With this approach, you can set certain global variables in your deployment scripts that apply to all your environments. This step might be more difficult if your development environment is based on Windows and your production environment is based on a version of UNIX. If you do not have a standardized location for installed applications, the second best approach might be to standardize on the default installation directory of each application simply to ease supportability.

## Application server

The application server is probably the most crucial. The application server plays a major role in IBM Content Navigator and the majority of the enterprise applications offered today. Consider standardizing the following names:

- Server names (server1)
- Profile names (AppSvr01)
- Node names (Node01) and cell names (Node01)
- Deployed application names (navigator1)

The reason for the suggestions is because implementing your environment by using the standardized names establishes predictability. For example, if you need a script to start and stop your server (recycle), you can write a script to loop a predefined number of servers (based on how many you have in a farmed or clustered environment).

And, predictably you can use the following example to invoke a start operation on each of the servers in the farm:

```
startServer.bat AppSvr$num
```

The point is that, even if you do not have a multiple-server development environment, you should resist the urge to use the name `server` to name the server; but instead name it `server1` to ensure that your scripts will work in multiple environments.

The same concepts apply to deployed application names. If you have multiple IBM Content Navigator instances installed in one application server, you can use a numbering scheme on which to base your scripts, so you can do quick application recycling. In addition, if you need a plug-in deployed in the plug-ins directory, then you might also need that plug-in for all instances, and you can use a similar script to cycle-copy the correct JAR files to the correct folders.

## Database

Maintaining a clean database is important. Many companies have stored procedures that help them accomplish administration and maintenance tasks. We advise creating a new tablespace and storing those and also helper and temporary tables there. This way, you do not have to give access to the IBM Content Navigator database to people who are promoting applications from environment to environment. Give them execute permissions to the stored procedures, which are in a separate tablespace, and make it so that their only entry point into persisting changes to the IBM Content Navigator database is those stored procedures. Standardizing on naming conventions from previous sections apply here also.

## Content repository

We focus on IBM FileNet Content Manager (P8). FileNet P8 is installed with numerous available classes that are ready to use. To preserve the integrity of the object store, we create all document classes from customers under a common super class. For example we create an abstract document class directly under Document (we are calling MyCompanyDocument), and create every customer document class as a subclass of MyCompanyDocument. We use this approach so that we can guarantee that the system-defined classes remain intact and that promoting classes always happens at the MyCompanyDocument level. Furthermore, if we have many object stores, we would make sure that wherever

documents are stored, we replicate the full document class hierarchy just so that we can manage changes easier in the future. For example, this technique not only makes multi-object-store searching easier in the future, but also allows us to migrate content from one object store to the other with minimal effort.

Again, remember to test and stage your deployments.

## 13.5 Using export and import to deploy a Content Navigator solution

Another approach for managing your deployment of an IBM Content Navigator deployment is to use the built-in export and import features to manage your application deployment from one environment to another. This approach is more seamless than a manual migration and eliminates most of the manual steps required in other approaches.

The export and import tools feature allows you to control the specific desktops definitions that you want to migrate and it supports the process through a configuration wizard where you can identify which components and subcomponents you want to migrate.

The export and import feature includes the identification of the desktop definition, repositories, plug-ins and other configured items associated with a particular desktop configuration. It does not include the export of user-associated items such as favorites and recent searches. And, it does not include the export or import of elements of your configuration that might be located outside the control of a particular IBM Content Navigator deployment, such as a plug-in JAR file.

Although the ability to cleanly import and export desktop definitions is provided as part of the IBM Content Navigator environment, understanding all the elements within your organization's application is important for your application management processes.

As mentioned previously in this chapter, extensive interdependencies might exist within different components of your system. Be sure to understand these elements and plan for their management during any migration or upgrade process.

To begin planning for moving or migrating your application, consider the following items to identify the interdependent elements within your source system and then plan for their creation or migration in your target system.

- Target Repository System: This target ECM repository is set up to match or mirror the configuration elements from your source system. Repository taxonomy such as Document Classes (FileNet Content Manager), Item Types (Content Manager 8), and Properties will not be migrated using the IBM Content Navigator Import or Export utility. Either manually create these elements or use repository tools to deploy or copy the source configuration.
- Plug-ins or support code: Any plug-in files that are referenced as part of the IBM Content Navigator Desktop Import or Export process must be copied from the source servers to the target servers before the configuration is imported to the target system.
- Image files or special graphics: Any image files that are used within the IBM Content Navigator Desktop must also be manually copied from the source server to the target server.

Understanding the elements that might not be collected as part of your Import and Export process will make migrating or moving your IBM Content Navigator environment more consistent and predictable.

### 13.5.1 Using the Export function within the Administration Desktop

The purpose of the desktop Export function with the IBM Content Navigator application is to allow you to move, migrate, or update one or more IBM Content Navigator applications. Figure 13-3 on page 483 shows the Export button that you click to start the export in the Content Navigator Desktops.

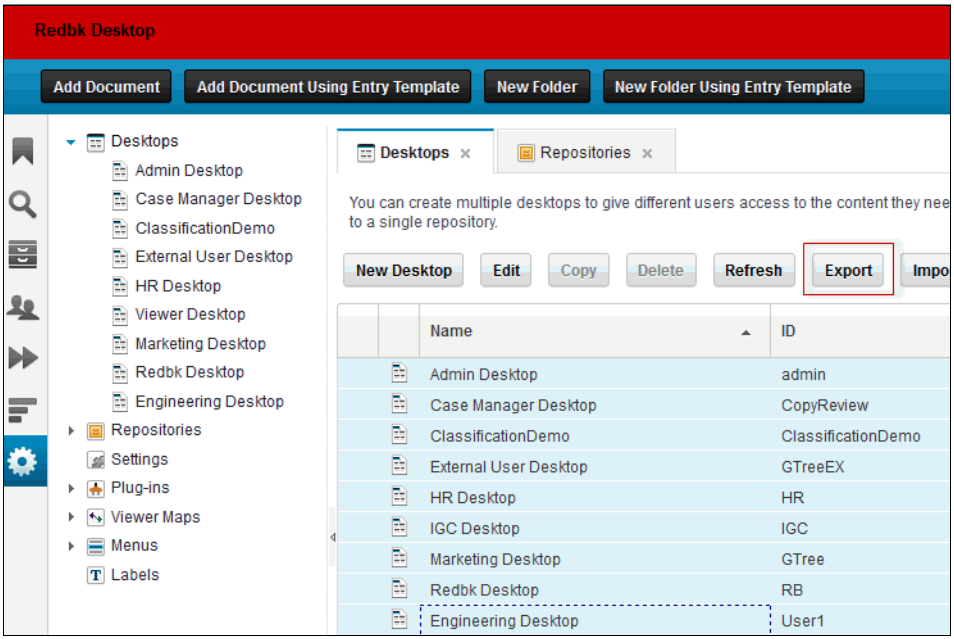



*Figure 13-3 The Export action button in the desktop*

## Desktops definition

To export a desktop definition, you must use the Administration view in the IBM Content Navigator Administration Desktop session. The Administration Desktop is the only desktop that allows you to export or import an IBM Content Navigator definition.

In the settings mode with **Desktops** selected, you see a list of all configured desktops in you IBM Content Navigator system.

The desktop definition is the first element that you select when exporting. You may select a single desktop definition in the list or select multiple definitions. After you select the desktop definitions that you want to export, select the Export function and the system opens the Export Desktop wizard. Figure 13-4 on page 484 shows the wizard with available options.

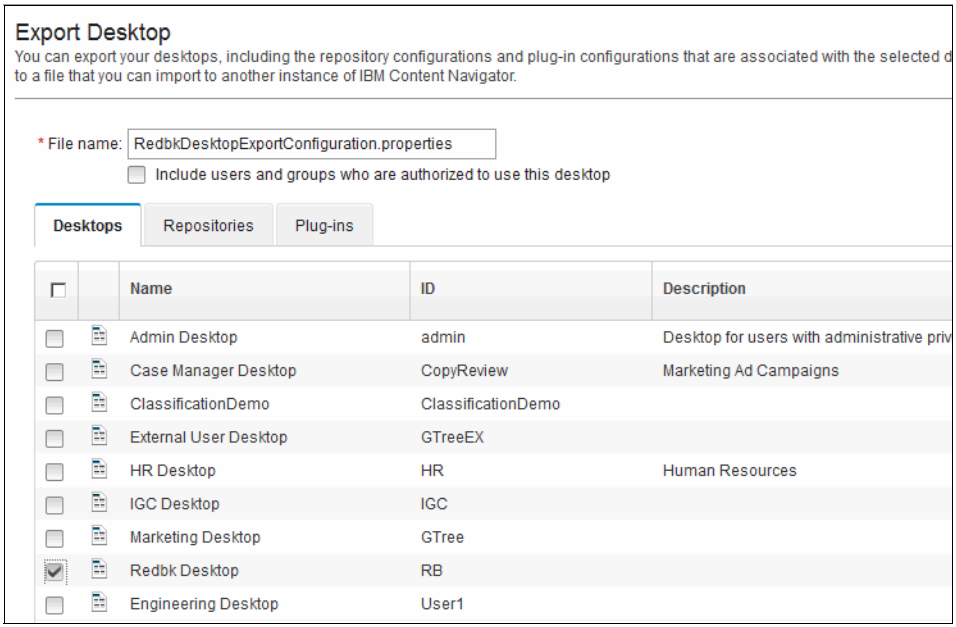



*Figure 13-4 Export Desktop tabs: Desktops*

The Export Desktop wizard has three tabs available:

- Desktops
- Repositories
- Plug-ins

## Desktops tab

The Desktops tab (Figure 13-4) is displayed by default when you open the Export Desktop wizard. You may again select the desktop definition you want to export. There is also a field for the file name of your export. It defaults to the `ExportedConfiguration.properties` file.

The Export Desktop wizard also provides a field where you can name your export file. The default file name is `ICNExportedConfiguration.properties`. You might want to name the export file by using a naming convention or in such a way that you can distinguish different export files from one another in the future.

If you select the **Include users and groups who are authorized to use this desktop** check box for the export, user or group definitions will not transfer between different LDAP environments. It identifies only the named user or group that is authorized for any desktop exports. Your organization is responsible for ensuring that the same user and group definitions are contained in separate

LDAP environments if they differ between your source and target IBM Content Navigator environments.

## Repositories tab

The Repositories tab in the wizard lists all available repositories that are connected to the desktop definitions that you selected on the Desktops tab page. Figure 13-5 shows the options in the Repositories tab.

Information about those repositories is available within this tab to help you identify the difference between the repositories selected. Display Name, Internal ID, Server Type, Server Name and Port Number are visible. These columns are not configurable (changeable) but can be sorted if you click a header column.

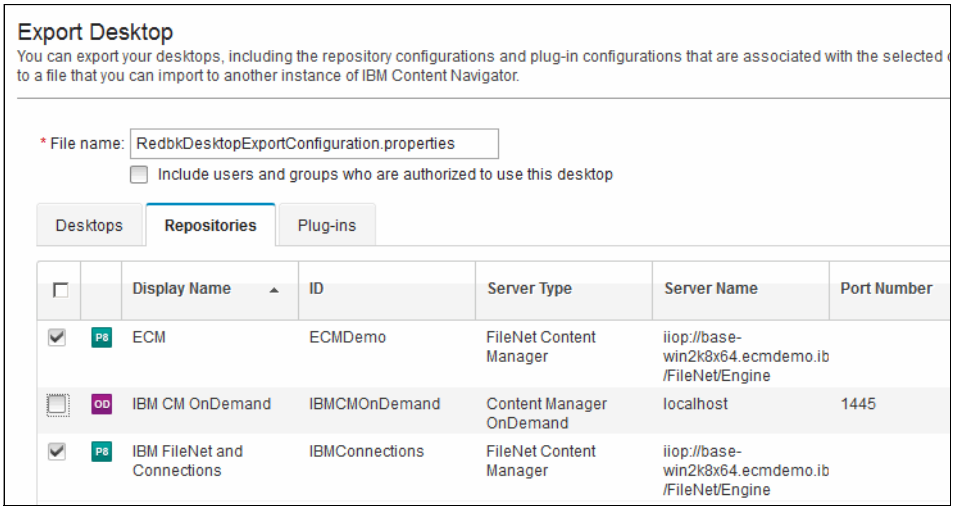



*Figure 13-5 Export Desktop tabs: Repositories*

## Plug-ins tab

The Plug-ins tab in the wizard shows all plug-ins that have been configured for the Desktop definitions that you selected in the Desktops tab. Figure 13-6 on page 486 shows the options available from the Plug-ins tab.

An important point to understand is that support for the export and import of plug-ins within a Desktop definition extends only to the configuration settings of that plug-in. The associated plug-in JAR files that might be deployed on your application server are not exported or imported as part of this process.

If you have plug-ins that you are using as part of your IBM Content Navigator application, you must manually copy and deploy the plug-in JAR file from your

source server to your target server so that the exported or imported plug-in configuration can work properly.

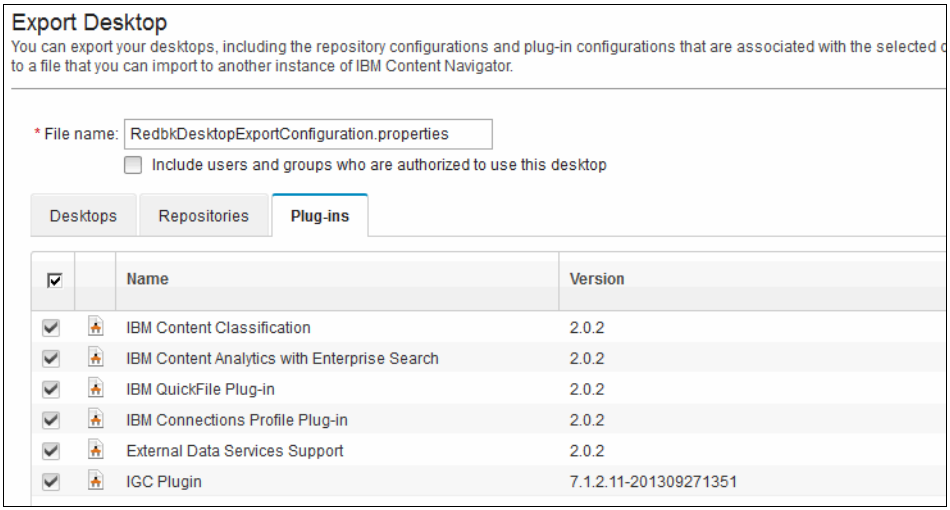



*Figure 13-6 Export Desktop tab: Plug-ins*

## Exporting the .properties file

After you complete the selections for desktops, repositories, and plug-ins, you can complete the export by clicking **Export** in the Export wizard. You are prompted to open or to save the file with the name you specified at the start of this operation. If you chose to save the file, it will be saved to the default downloads location on your local system. Figure 13-7 shows an example of a saved `.properties` file:

`RedbkDesktopExportConfiguration.properties`

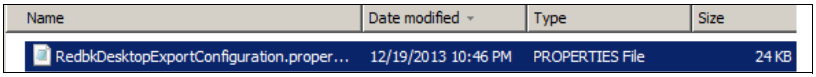



*Figure 13-7 Export file*

Example 13-3 shows a truncated portion of the contents of that file.

*Example 13-3 Export configuration properties file contents*

```
menu.navigator.DefaultICAPluginItemContextMenu =
pluginId=ICAPlugin;typeLabel=ICA Plugin Item Context Menu
Type;name=ICAPlugin Item Context
Menu;type=ICAPluginItemContextMenu;description=ICA Plugin context menu
for search results menu.navigator.DefaultICAPluginECMItemContextMenu =
pluginId=ICAPlugin;typeLabel=ICA Plugin ECM Item Context Menu
```

```
Type;name=ICAPlugin ECM Item Context
Menu;type=ICAPluginECMItemContextMenu;description=ICA Plugin context
menu for Enterprise Content Management results
interfaceText.navigator.RB.GTreeEX.GTree.CopyReview.favorites =
labelKey=Favorites;displayedIn=desktop;type=desktop;desktopId=RB
interfaceText.navigator.RB.GTreeEX.GTree.CopyReview.browse =
labelKey=Browse;displayedIn=desktop;type=desktop;desktopId=RB
interfaceText.navigator.RB.GTreeEX.GTree.CopyReview.search =
labelKey=Search;displayedIn=desktop;type=desktop;desktopId=RB
interfaceText.navigator.RB.GTreeEX.GTree.CopyReview.casesearch =
labelKey=Case Search;displayedIn=desktop;type=desktop;desktopId=RB
interfaceText.navigator.RB.GTreeEX.GTree.CopyReview.datacap =
labelKey=Datacap;displayedIn=desktop;type=desktop;desktopId=RB
mobilefeature.navigator.RB.GTreeEX.GTree.CopyReview.favorites =
iconFile=/navigator/ecm/widget/resources/images/mobileFavorites.png;url
=;name=Favorites;desktopId=RB;display=false
mobilefeature.navigator.RB.GTreeEX.GTree.CopyReview.browse =
iconFile=/navigator/ecm/widget/resources/images/mobileBrowse.png;url=;n
ame=Browse;desktopId=RB;display=false
mobilefeature.navigator.RB.GTreeEX.GTree.CopyReview.search =
iconFile=/navigator/ecm/widget/resources/images/mobileSearch.png;url=;n
ame=Search;desktopId=RB;display=true
mobilefeature.navigator.RB.GTreeEX.GTree.CopyReview.casesearch =
iconFile=/navigator/ecm/widget/resources/images/mobileCase.png;url=;nam
e=Case Search;desktopId=RB;display=false
mobilefeature.navigator.RB.GTreeEX.GTree.CopyReview.datacap =
iconFile=/navigator/ecm/widget/resources/images/mobileDatacap.png;url=;
name=Datacap;desktopId=RB;display=false desktop.navigator.RB =
isDefault=No;bannerMenuColor=#FFFFFF;TeamspacesListToolbar=DefaultTeams
pacesListToolbar;TemplatesListToolbar=DefaultTemplatesListToolbar;helpU
rl=;description=;BannerUserSessionContextMenu=DefaultBannerUserSessionC
ontextMenu;FavoriteItemContextMenu=DefaultFavoriteItemContextMenu;passw
ordRulesUrl=;mobileAppAccess=true;promptCloseOffice=true;ObjectStoreFol
derContextMenu=DefaultObjectStoreFolderContextMenu;UserQueueContextMenu
=DefaultUserQueueContextMenu;accessControlEnabled=false;AttachmentItemC
ontextMenu=DefaultAttachmentItemContextMenu;SelectObjectToolbar=Default
SelectObjectToolbar;loginLogoUrl=http://base-win2k8x64:9081/navigator/e
cm/widget/resources/images/redbk.png;
```

## 13.5.2 Using the Import function within the Administration Desktop

The Import Desktop wizard has five tabs:

- Desktops tab
- Repositories tab
- Plug-ins tab
- Menus tab
- Labels tab
- Mobile Features tab

The Desktops tab opens in the wizard by default (Figure 13-8); you may begin by selecting the desktop definitions you want to export.

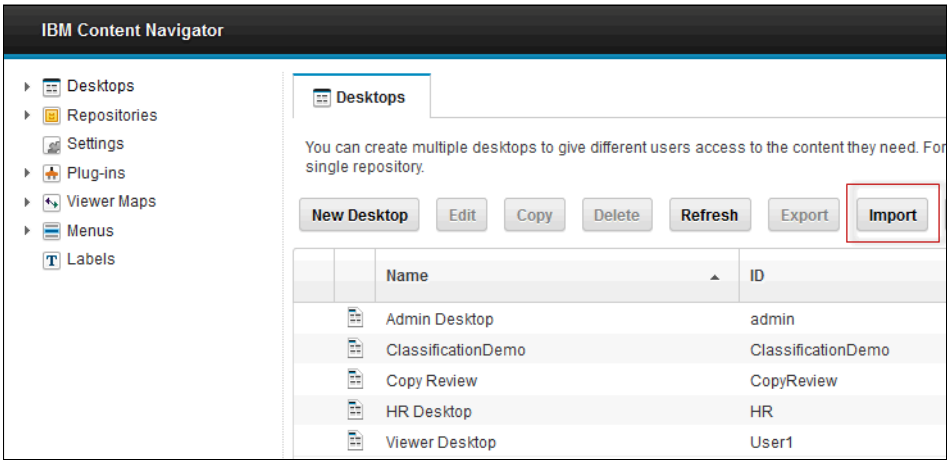



*Figure 13-8 Desktop Actions Menu: Import*

### Desktops tab

To import a desktop definition, you must open the Administration view within the IBM Content Navigator Administration Desktop session. The Administration Desktop is the only desktop that will allow you to export or import an IBM Content Navigator Definition.

In the Administration view, the Import option is available from the function menu. Click **Import**. You are prompted to supply or select the location of the desktop definition file you want to import. This file is the `.properties` file that you created, named, and saved when you exported it (see “Exporting the .properties file” on page 486).

If, during the import, an element that is in the import file already exists in the system, a warning message is displayed at the top of the Import Desktop window. The default behavior for any items in conflict is to not update the item. As

a result, the conflict item check boxes are not selected in the list of items to import, as shown in Figure 13-9. If you want to import or overwrite, select the conflict items.

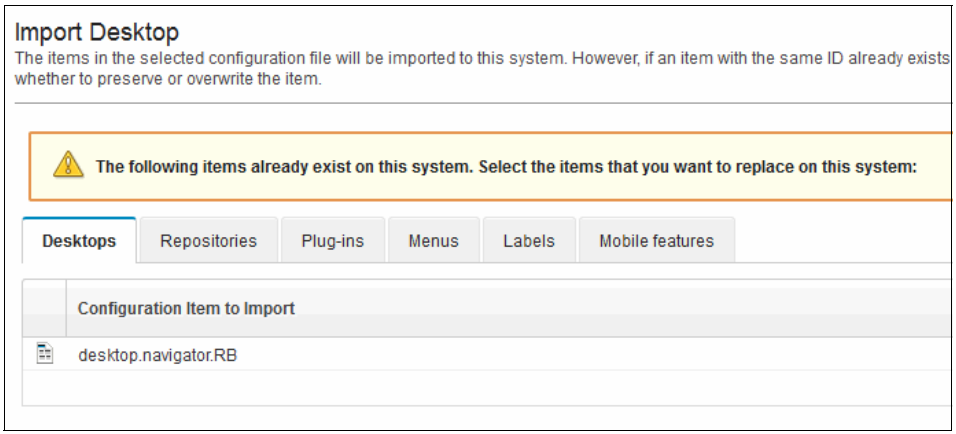



*Figure 13-9 Import Desktop tabs: Desktops, showing conflicting items for import*

## Repositories tab

The Repositories tab (Figure 13-10) in the wizard lists all available repositories that are defined within the export or import file.

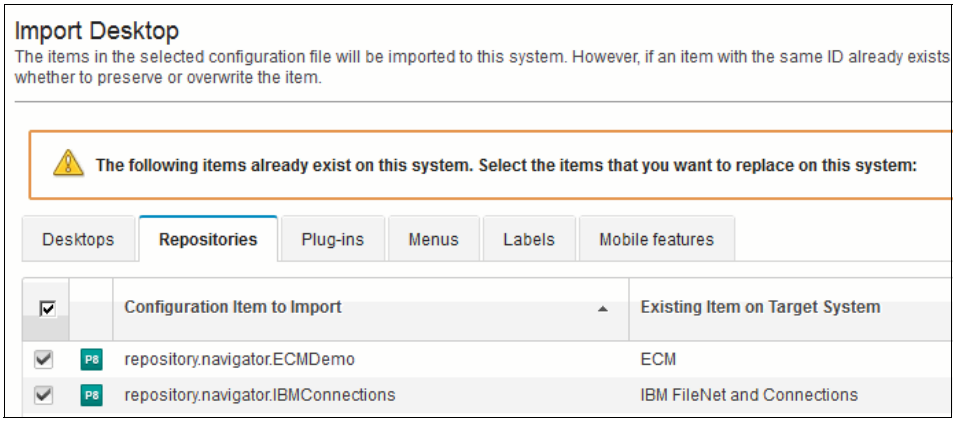



*Figure 13-10 Import Desktop tabs: Repositories*

## Plug-ins tab

The Plug-ins tab (Figure 13-11) in the wizard shows all plug-in configurations that have been exported for the Desktop definitions that you selected previously.

Support for exporting and importing of plug-ins within a Desktop definition extends only to the configuration settings of that plug-in. The associated plug-in JAR files that might be specified within the configuration of your import file must be deployed on your application server so that the imported configuration can work properly.

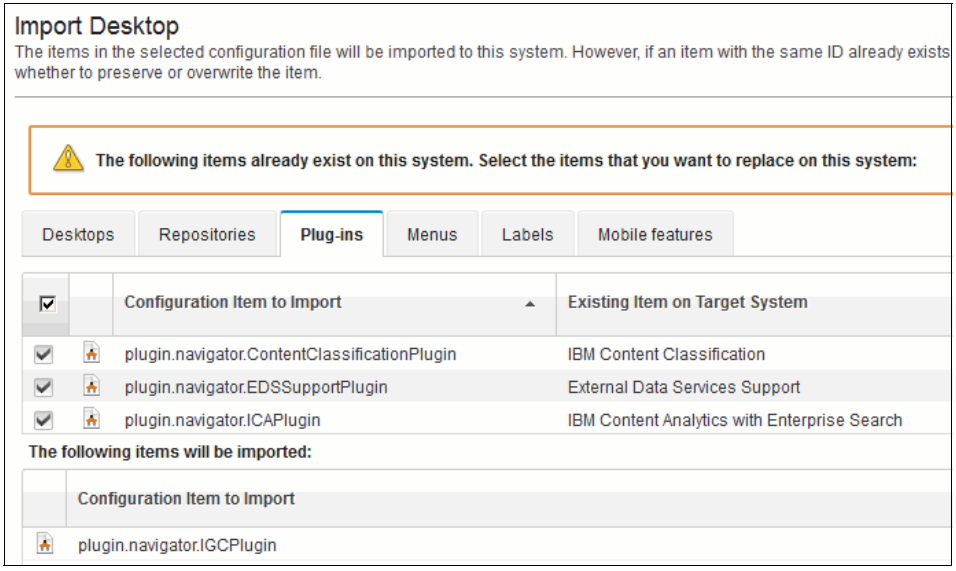



*Figure 13-11 Import Desktop tabs: Plug-ins*

## Menus tab

The Menus tab (Figure 13-12) in the wizard shows all menus available for import.

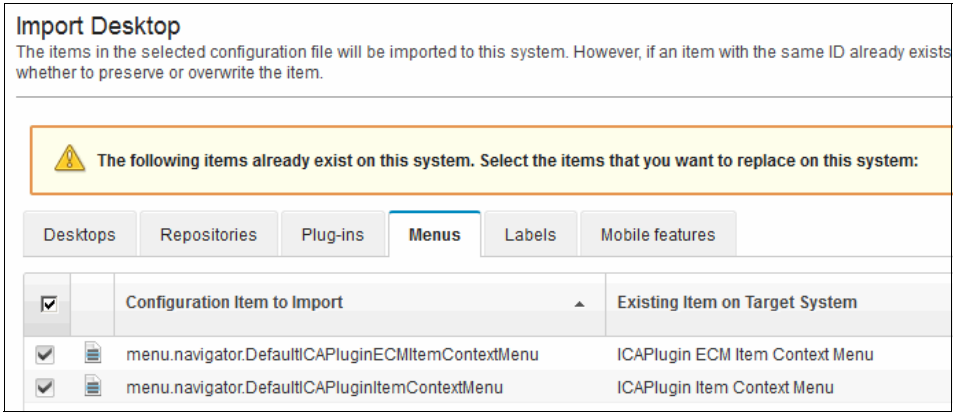



*Figure 13-12 Import Desktop tabs: Menus*

## Labels tab

The Labels tab (Figure 13-13) in the wizard lists the labels configurations that are available to import to your system.

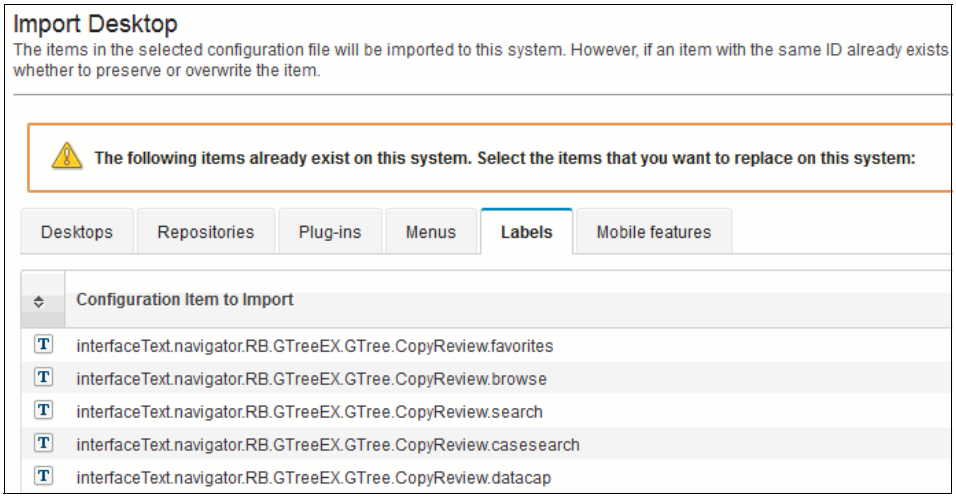



*Figure 13-13 Import Desktop tabs: Labels*

### Mobile Features tab

The Mobile Features tab (Figure 13-14) in the wizard shows the mobile features that are available to import to your system.

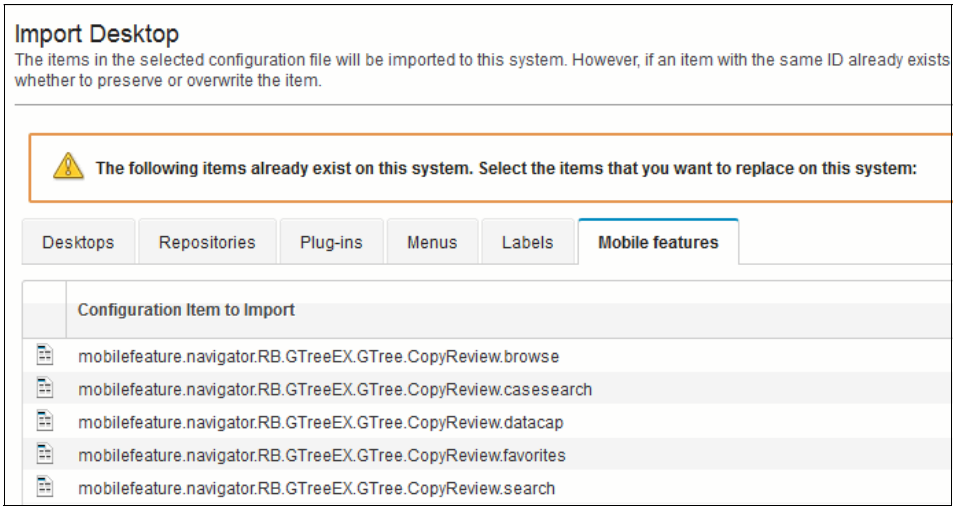



*Figure 13-14 Import Desktop tabs: Mobile Features*

## 13.5.3 Import summary and reports

The import action includes a summary report after the import is complete. This summary includes a list of each element type (desktops, repositories, plug-ins, labels, and mobile features) and provides information of how many of each of those elements were imported and how many were skipped during the operation. The bottom of the summary report shows an action available to download the full import report. That report will contain import status information for each individual element that was available in the export or import file used.

Figure 13-15 shows an example.

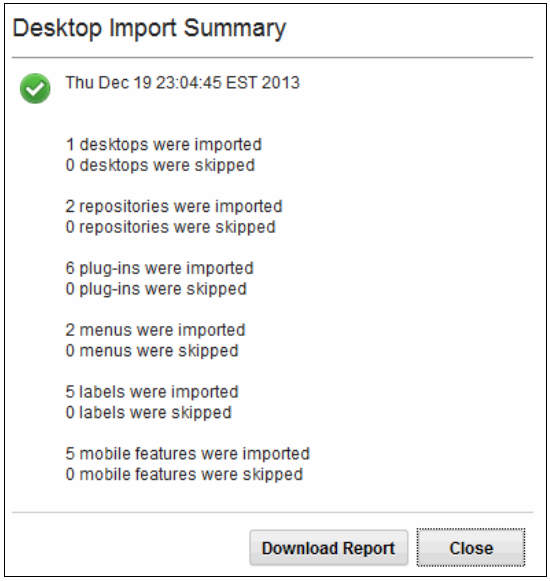



*Figure 13-15 Desktop Import Summary and Download Report*

## 13.6 Conclusion

This chapter explores strategies for managing your production environment as it applies to IBM Content Navigator solutions. The chapter offers an approach to managing the components of your application. It also provides an overview to establishing a level of predictability when managing your environments and moving or promoting your solutions between various environments.

Although traditional approaches to migrating your solutions, such as manual solution deployment, provide a valid process to managing your enterprise environments, the IBM Content Navigator Administration Desktop provides tools and support to simplify solution deployment and migration.

With any production solution, a preferable approach is to automate deployment as much as possible to provide consistency and predictability in your enterprise environment. IBM Content Navigator includes a simplified approach to moving, migrating, or promoting your desktop solutions.

14

# Debugging and troubleshooting

A well-planned troubleshooting strategy in an application can reduce the time to identify and fix errors, and can ease the learning curve in extending software packages. This chapter describes troubleshooting techniques that can be used when you customize and extend IBM Content Navigator. We present various log files and offer tips for troubleshooting.

Troubleshooting is a process of gathering information, analyzing, identifying actions based on analysis, and implementing remedial actions. The main focus of this chapter is to identify the information that is important and how to gather it for troubleshooting purpose. Use the information presented in this chapter in conjunction with IBM Content Navigator online documentation for support and troubleshooting.

This chapter covers the following topics:

- Client debugging
- Client logging
- Server-side logging
- Log and trace files
- Troubleshooting

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# 14.1 Client debugging

When you develop an extension for IBM Content Navigator, the choice of integrated development environment (IDE) and the choice of runtime execution can help the development and test process.

Chapter 3, “Setting up the development environment” on page 73 describes how to set up a development environment by using either IBM Rational Application Developer or the Eclipse IDE with web tools plug-in.

To test the individual plug-in component, the plug-in must be registered and enabled by selecting it for use in a desktop profile.

When you test a client module, use the developer tools option of your browser to log in to the console. See 14.2, “Client logging” on page 498 for details of browser console logging. To test the server-side or business logic of your component, see 14.3, “Server-side logging” on page 500.

There are many tools can be used for web application client debugging. We introduce several in the following sections.

## 14.1.1 Firefox

Firebug is a powerful client debugging tool for Firefox. After you install this add-on, press F12 to launch it in the browser. The Firebug option can be set to any location of the browser or separately in an individual window.

Firebug has several panels that are used for various purposes:

- Console: Show logs and all requests and response content.
- HTML: Shows page HTML source.
- CSS: Shows CSS of current page.
- Script: Shows scripts. You can debug JavaScript code here. You can set breakpoints, watch variable values, and view the whole stack.
- DOM: Shows DOM information.
- Net: Shows network transmission information, such as URL, status, size, and timeline.
- Cookie: Shows all cookie information.

For more information about Firebug, see the following web page:

`http://addons.mozilla.org/en-US/firefox/addon/firebug`

### 14.1.2 Chrome

Chrome has various integrated developer tools. Press F12 to see them. Clicking the **Unlock into separate window** button can separate it as an individual window.

In the Chrome tool, several panels offer various functions and information:

- Elements: Shows the whole structure of a web page. All DOM nodes can be viewed here.
- Network: Shows network transmission information.
- Sources: Shows scripts. You can debug JavaScript code here. You can also modify the JavaScript in this panel; Chrome will then run the modified version. This is convenient for your development.
- Timeline: Shows a performance timeline.
- Profiles: Shows profiles.
- Resources: Shows all resources used in the current page.
- Audits: Runs audits of the code that you select. It provides advice regarding potential issues.
- Console: Shows logs. No request and response content is shown in the console.

For more information about Chrome, see the following website:

`http://www.google.com/chrome`

### 14.1.3 Fiddler

The full name is Fiddler Web Debugger. It can monitor all requests and responses of browsers. You can set Fiddler to monitor only one browser process. A function panel provides details of request, response, and network transmission.

There is a JSON viewer plug-in for Fiddler. It can view JSON data in a structured tree. For IBM Content Navigator, all JSON responses have the following header:

```
{}&&
```

To view JSON in a JSON viewer, this heading must be removed. Fiddler can be run in a stand-alone mode to view only the JSON string.

For more information about fiddler, see the following website:

`http://fiddler2.com`

## 14.2 Client logging

All browsers display logged messages to the browser console. By default, all JavaScript and system errors are logged. All browsers support the opening of JavaScript files to find the source of an error. However, sometimes this type of logging is not sufficient to help resolve errors that are related to data or logic.

### 14.2.1 Logging levels and browser types

IBM Content Navigator provides the `ecm.logger` logging class for use in developing plug-ins. The logging class is used to selectively log messages by five levels:

- Level 0: None (This level indicates no logging.)
- Level 1: Error
- Level 2: Warning (This level is the default logging level.)
- Level 3: Information
- Level 4: Debug

To invoke the desktop with the debug parameter, use the following line:

`http://<server_name>:<port>/navigator/?debug=true`

It has the following meanings:

- *<server_name>* is the DNS resolvable name of the web application server that hosts the navigation web client application.
- *<port>* is the port number for the application server. The default for WebSphere Application server is 9080.

**Performance:** Logging debug messages affects performance. However, later versions of browsers are generally more efficient so this option has less effect on load times when used with newer versions of browsers.

To view the console in your web browser, press the F12 function key. In the Safari browser, however, the F12 key is the shortcut to invoke the JavaScript console. If you use Safari, press Ctrl+Alt+C.

If you use Firefox and do not have a debug tool, such as Firebug, installed, the log messages can be redirected to a pop-up window by using the following command:

`http://<server_name>:<port>/navigator?debug=true&useConsole=false`

Using the `debug=true` parameter is the equivalent of setting the logging level to 4. If you want to reduce the logging level, use the logLevel parameter. For example, to log errors use the following URL:

`http://<server_name>:<port>/navigator?logLevel=1&useConsole=false`

Errors can now be logged to a pop-up window.

## 14.2.2 Enabling logging in JavaScript files

To enable the ECM logger in your JavaScript, include the following statement:

```
dojo.require("ecm.LoggerMixin");
```

If you use AMD loading, include the following statement:

```
ready(["ecm/LoggingMixin","your required widgets",
function yourFunction(){
   // enter your code here
});
```

The ecm.LoggerMixin class provides an ECM logger class and provides functions for logging the various message levels. To log a message using one of the functions provided by LoggerMixin class, use the following format:

```
this.logInfo(yourfunctionname, message, extra);
```

LoggerMixin provides the following functions:

- logInfo: Logs an information message.
- logWarning: Logs a warning message.
- logError: Logs an error message.
- logDebug: Logs a debug message.

Two additional message types can be used:

- logEntry (Function Name, Message): Logs the entry point to your function.
- logExit (Function Name, Message): Logs the exit point from your function.

The entry and exit logging functions use only two parameters.

**Tips:** All supported browsers include debugging tools. If you are new to debugging JavaScript code, each browser offers online help regarding the use of the developer tools. See the appropriate browser help pages for information about use of developer tools for debugging client code.

## 14.3 Server-side logging

Chapter 3, “Setting up the development environment” on page 73 describes how to enable server-side debugging techniques for both Rational Application Developer for WebSphere and for the Eclipse development environment.

One aspect of debugging is to use logging to record and trace the execution path of your code. For a description of how to start client logging, see 14.2, “Client logging” on page 498. If you want to log server execution, it must be enabled by using the administration feature of the desktop or by using the Administration Desktop.

The effect of client-level logging and the suggested logging at the appropriate level is described in 14.2, “Client logging” on page 498. The effect of client-level logging is specific to the individual desktop that is being logged. The server-side logging, however, might affect all users and so the best approach is to minimize the effect when possible.

This section describes how to enable logging for a plug-in example, showing techniques to minimize the effect on the server.

**Note:** The focus in this chapter is on the use of IBM software. If you use WebLogic as the application server, the system log file has a slightly different format and path. For a description of the WebLogic server log file, go to the following address:

`http://docs.oracle.com/cd/E13222_01/wls/docs81/ConsoleHelp/logging.html#1032324`

Although the WebLogic log format differs slightly, the log message from IBM Content Navigator contains the same information as described in 14.3, “Server-side logging” on page 500.

We use server logging to trace the execution of the main Java class, CustomSearchPlugin, of an example plug-in.

To enable server-side logging in IBM Content Navigator, use the following steps:

1. Connect to IBM Content Navigator as an administrator, or use the Administration Desktop.

1. Select **Settings** from the Feature panel.
2. Select the **Logging** tab.
3. Enter `Debug` for the Application-level logging.

4. Select **Class-level Logging Specific Classes**.
5. Enter the following information:

   ```
   com.ibm.ecm.extension.customsearch.*
   ```

Logging can be enabled for specific users or specific client devices. Use this option if you need to use debug on a production instance of IBM Content Navigator. Logging can also be enabled for specific classes. Use this option to debug your classes during development. You can specify your classes and optionally exclude specific classes.

Restricting the scope of logging reduces the effect on performance and aids troubleshooting and debugging.

**Tips:** To minimize the server-side load of logging, enable logging for specific classes. If you enable logging for all classes, you can reduce the effect in the following ways:

- Exclude classes; enter a comma-delimited list of classes to exclude from the logging list.
- Log only for specific user workstations by entering the IP address of the client workstation that will be logged.
- Enable logging for specific users.

Figure 14-1 on page 502 shows the server logging setup for a plug-in class. The example shows the restricting of logging to one class and for one specific IP address.

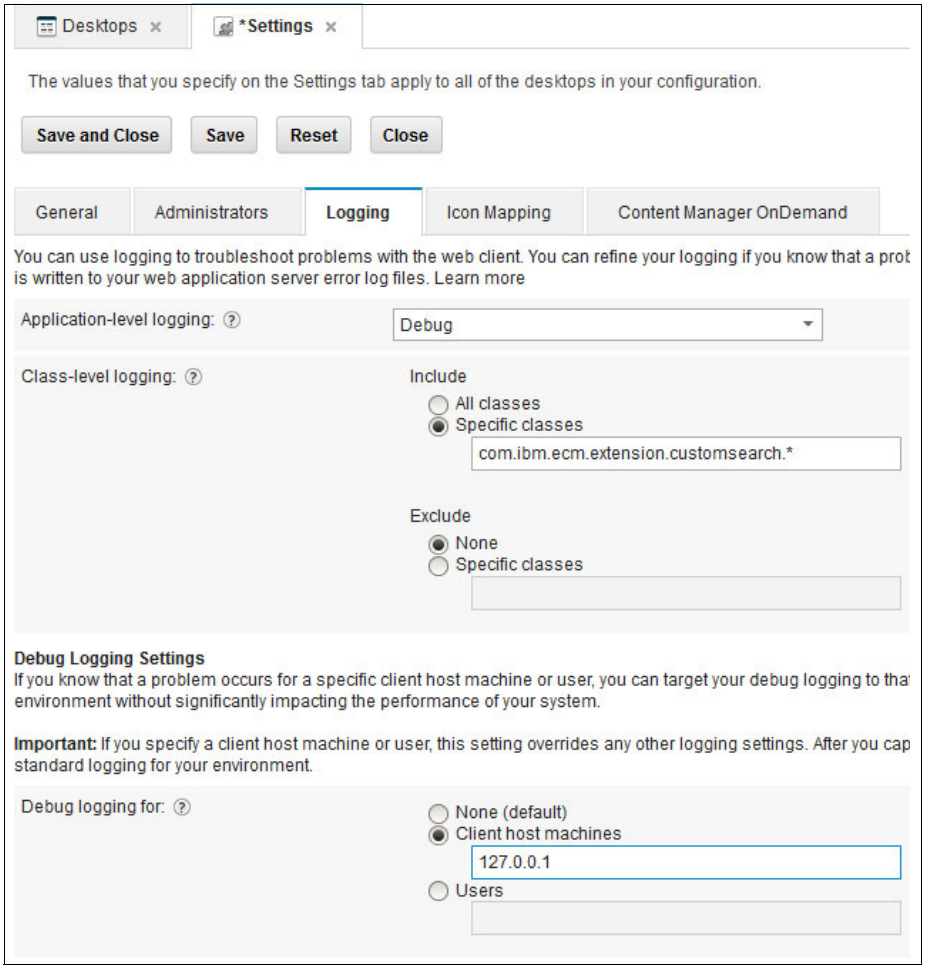



*Figure 14-1 Enable server logging to a specific class and a specific client machine*

Server logging is written to the web application server log file. For example the WebSphere Application Server, Version 7 log file, `SystemOut.log`, is in the following location:

```
C:\Program\Files\IBM\WebSphere\AppServer\profiles\AppSrv01\logs\server1
```

Information is logged as a series of requests that are delimited by these lines, where `nnn` is a numeric value, logged in ascending order:

```
Begin Request nnn
End Request nnn
```

Example 14-1 is an extract of the SystemOut.log file from an IBM Content Navigator system, hosted in WebSphere Application Server Version 7. It shows the execution path of the com.ibm.ecm.extension.customsearch.* Java classes.

*Example 14-1 Extract of SystemOut.log with logging enabled for example plug-in*

```
[10/30/13 16:46:44:194 CST] 00000012 SystemOut  O
CIWEB.CustomSearchPlugin Entry: [Administrator @ 127.0.0.1]
com.ibm.ecm.extension.customsearch.SearchService.execute()
[10/30/13 16:46:44:195 CST] 00000012 SystemOut  O CIWEB Entry:
[Administrator @ 127.0.0.1]
com.ibm.ecm.extension.customsearch.SamplePluginSearchServiceP8.executeP
8Search() Query String: Select * from DOCUMENT  where  this INFOLDER
{B3BB3B22-EB21-4997-ACF8-87DAE91966AA} OPTIONS (COUNT_LIMIT 2147483647)
[10/30/13 16:46:44:256 CST] 00000012 SystemOut  O
CIWEB.CustomSearchPlugin Exit : [Administrator @ 127.0.0.1]
com.ibm.ecm.extension.customsearch.SearchService.execute()
```

The first entry logs the details of the search service launched by clicking folder node of the custom tree. The log entry is formatted in the following way, as shown in Example 14-2:

*[date time tz] thread stream CIWEB loggingLevel: [userId @ hostNameOrIP] package.class.method() message*

*Example 14-2 Log entry message*

```
[10/30/13 16:46:44:195 CST] 00000012 SystemOut  O CIWEB Entry:
[Administrator @ 127.0.0.1]
com.ibm.ecm.extension.customsearch.SamplePluginSearchServiceP8.executeP
8Search() Query String: Select * from DOCUMENT  where  this INFOLDER
{B3BB3B22-EB21-4997-ACF8-87DAE91966AA} OPTIONS (COUNT_LIMIT 2147483647)
```

The logged message shows that at 16:46 on 30 October 2013, thread 00000012 used the following line:

```
com.ibm.ecm.extension.customsearch.SamplePluginSearchServiceP8.executeP8Search
```

It used that line to do the custom search against IBM FileNet Content Manager with the following query string:

```
Select * from DOCUMENT  where  this INFOLDER
{B3BB3B22-EB21-4997-ACF8-87DAE91966AA} OPTIONS (COUNT_LIMIT 2147483647)
```

Reading the log entries shows the IBM Content Navigator functions that are executed, the parameters used, and any messages that are returned. With this information, identifying a failing component and the primary symptom are possible.

The log file will contain many entries that are tracing the execution of the logged classes. Use a text editor to browse and search for information. Make a copy of the log file so that you can compare results after taking the necessary actions.

## 14.4 Log and trace files

In addition to the console log, described in 14.2, “Client logging” on page 498, and the system log, described in 14.3, “Server-side logging” on page 500, several other log files are important sources of information.

### 14.4.1 IBM Content Navigator log files

Review the following files when you troubleshoot. Also, provide them to the IBM Support group if you log a problem (PMR). The IBM Support Assistance Data Collector, described in 14.5, “Troubleshooting” on page 505, automatically collects the IBM Content Navigator log files.

- Installation log file
- Version file
- Deployment profile
- Temporary files
- Configuration files

See the IBM Content Navigator at the IBM Knowledge Center by using one of the following addresses.

- For use with IBM Content Manager:

  `http://publib.boulder.ibm.com/infocenter/cmgmt/v8r5m0`

- For IBM Content Manager OnDemand:

  `http://publib.boulder.ibm.com/infocenter/cmod/v9r0m0`

- For IBM FileNet Content Manager:

  `http://publib.boulder.ibm.com/infocenter/p8docs/v5r2m0`

The IBM Content Navigator client logs to the Java console of the browser. By default, this information is not written to disk. Therefore, copying the contents of the log file to a text file for analysis and review is important.

IBM Content Navigator provides viewing capability for a variety of file types. To debug viewer-specific issues, review the log file. FileNet Viewer includes a log file in either of the following locations:

- `/opt/IBM/FileNet/WebClient/ImageViewerPro/streamer`
- `C:\Program Files(x86)IBM\ECMClient\config\streamer.log`

This file contains information about any errors that stream data from an IBM FileNet Content Manager repository.

### 14.4.2 Application server log files

IBM Content Navigator logs to the following file for WebSphere Application Server:

```
c:\Program Files\IBM\WebSphere\Appserver\profiles\Appsrvr01\logs\System.out
```

For WebLogic, IBM Content Navigator logs to the following file:

```
C:\mydomain\servers\myserver\logs\myserver.log
```

WebLogic transfers log entries to a domain log file for all instances of WebLogic servers. By default, the debug messages are not transferred.

## 14.5 Troubleshooting

The IBM Knowledge Center for IBM Content Navigator contains a structured approach to troubleshooting errors that can occur when installing, configuring, deploying, and running IBM Content Navigator. See 14.4.1, “IBM Content Navigator log files” on page 504 for links to the information center.

The IBM Knowledge Center is located within the information center for each content repository. Each information center has troubleshooting information that is specific to the repository. The information addresses actions for a known set of issues at the time it is published. IBM Support also maintains a knowledge base of reported current problems and resolutions. Use both the information center and the knowledge base when you troubleshoot issues.

To search the knowledge base, use the following steps:

1. Go to the IBM Support Portal:

   http://ibm.com/SupportPortal
2. Search with Content Navigator in the Product lookup field.
3. Select the **Troubleshooting documentation** link.

Figure 14-2 shows the IBM Support troubleshooting window for IBM Content Navigator.

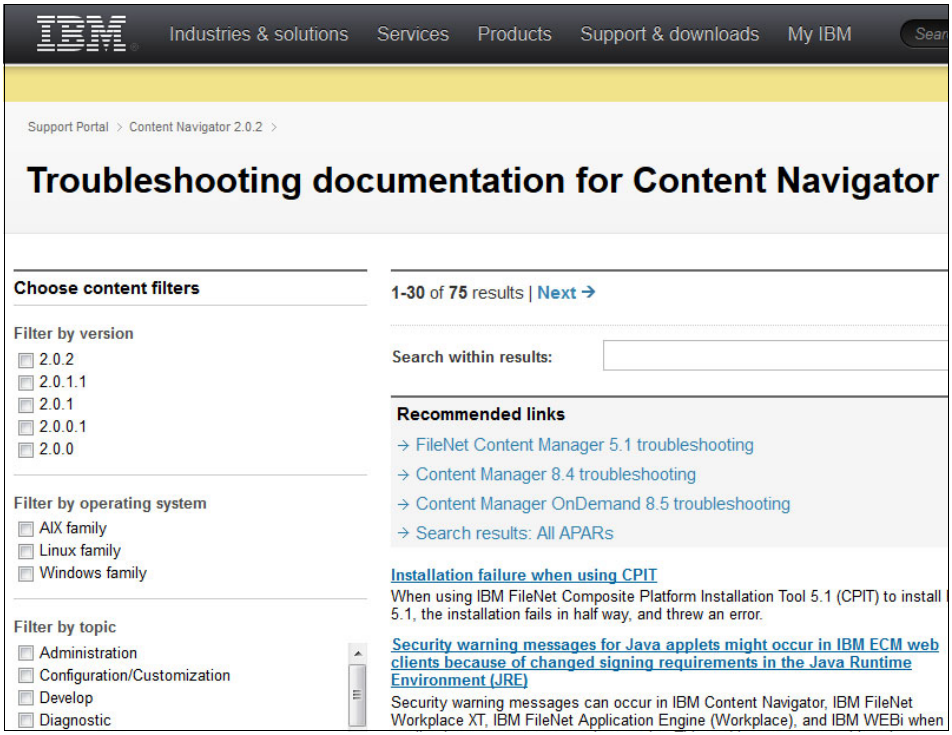



*Figure 14-2 IBM Support Portal*

## 14.5.1 IBM Content Navigator troubleshooting tools

IBM Content Navigator contains three tools that can help you troubleshoot or report a problem to IBM:

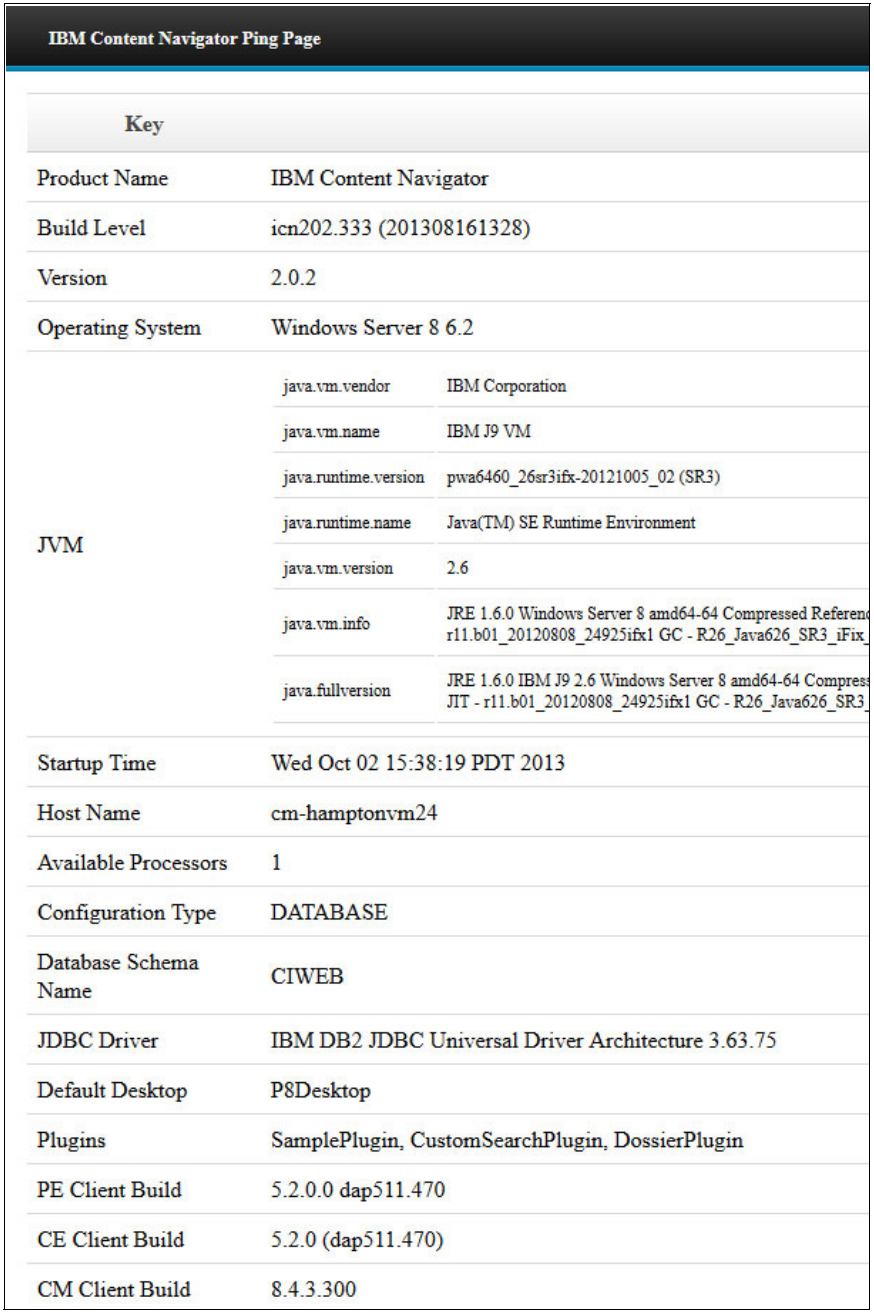
- IBM Content Navigator ping page
- Viewer configuration verify
- Configuration export

### IBM Content Navigator Ping Page

IBM Content Navigator contains a ping page that displays information about the installed software and the current status. To display the ping page, enter the following URL in your web browser:

`http://<server_name>:<port>/navigator/Ping`

Figure 14-3 shows the result of the ping page.



**IBM Content Navigator Ping Page**

| Key | | |
|---|---|---|
| Product Name | IBM Content Navigator | |
| Build Level | icn202.333 (201308161328) | |
| Version | 2.0.2 | |
| Operating System | Windows Server 8 6.2 | |
| JVM | java.vm.vendor | IBM Corporation |
| | java.vm.name | IBM J9 VM |
| | java.runtime.version | pwa6460_26sr3ifx-20121005_02 (SR3) |
| | java.runtime.name | Java(TM) SE Runtime Environment |
| | java.vm.version | 2.6 |
| | java.vm.info | JRE 1.6.0 Windows Server 8 amd64-64 Compressed Referenc<br>r11.b01_20120808_24925ifx1 GC - R26_Java626_SR3_iFix_ |
| | java.fullversion | JRE 1.6.0 IBM J9 2.6 Windows Server 8 amd64-64 Compress<br>JIT - r11.b01_20120808_24925ifx1 GC - R26_Java626_SR3_ |
| Startup Time | Wed Oct 02 15:38:19 PDT 2013 | |
| Host Name | cm-hamptonvm24 | |
| Available Processors | 1 | |
| Configuration Type | DATABASE | |
| Database Schema Name | CIWEB | |
| JDBC Driver | IBM DB2 JDBC Universal Driver Architecture 3.63.75 | |
| Default Desktop | P8Desktop | |
| Plugins | SamplePlugin, CustomSearchPlugin, DossierPlugin | |
| PE Client Build | 5.2.0.0 dap511.470 | |
| CE Client Build | 5.2.0 (dap511.470) | |
| CM Client Build | 8.4.3.300 | |

*Figure 14-3 IBM Content Navigator Ping Page*

## Viewer Configuration Verify

IBM Content Navigator provides a rich set of viewing capability. To check the configuration of the various viewing options. use the following web page:

`http://<server_name>:<port>/navigator/viewers/verify.jsp`

Example 14-3 shows the content of `verify.jsp` page.

*Example 14-3 The /navigator/viewers/verify.jsp page*

```
<%@page import="java.io.File, java.util.Properties,
java.io.FileInputStream, java.io.PrintWriter,
javax.xml.transform.TransformerFactory,com.ibm.ecm.configuration.*,com.
ibm.ecm.*,com.ibm.ecm.extension.Plugin,com.ibm.ecm.util.PluginUtil,
java.util.*" %>
<%
boolean ENABLE_VERIFY = false;
// To enable this page, uncomment the line below:
ENABLE_VERIFY = true;
//
```

The `verify.jsp` page checks the presence and validity of the IBM Content Navigator views that are installed on the system. Make sure to remove the comment from the following line before you run it.

```
ENABLE_VERIFY = true
```

A viewer verification page opens, similar to Figure 14-4 on page 509.

After verifying, be sure to disable the page by reapplying the comment for `ENABLE_VERIFY = true`.

## Verify ViewONE Pro

| Setting | Value | Validity |
|---|---|---|
| iviewproConfigurationDirectory (in web.xml) | C:/Nexus/daeja/config | Ok |
| Streamer configuration file | C:\Nexus\daeja\config\iviewpro.streamer.properties | Ok |
| Streamer cache directory | C:/projects/nexus/workspace/daeja/cache | Ok |
| Streamer log directory | C:\projects\nexus\workspace\daeja\logs | Ok |
| Streamer log file | C:/projects/nexus/workspace/daeja/logs/streamer.log | Ok |
| Redact configuration file | C:\Nexus\daeja\config\iviewpro.redacttofile.properties | Ok |
| Redact log directory | C:\projects\nexus\workspace\daeja\logs | Ok |
| Redact log file | C:/projects/nexus/workspace/daeja/logs/redacttofile.log | Ok |

## Verify XSLT

XSLT verification passed.

## Verify AFP Transform Configuration

Using Configuration Type: PROPERTIES

| OD Repository Id | OD Repository Name | AFP2PDF Available |
|---|---|---|
| grnvm139 | grnvm139 (OD 8.5.0.1) | Yes |
| cmi149 | OD Server cmi149 | afp2pdf has not been configured for this repository. |
| cm-hamptonvm15 | cm-hamptonvm15 (OD 8.5.0.1) | Yes |

## Verify OutsideIn Install

**PATH:** C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerS
Files\IBM\Infoprint Select;C:\Program Files (x86)\ATI Technologies\ATI.ACE\Core-Static;C:\Program Files
(x86)\IBM\SDP\jdk\bin;c:\bin;c:\projects\nexus\workspace\navigator\build\lib\inso\Win32;C:\projects\nexus\workspa

OutsideIn was found in c:\projects\nexus\workspace\navigator\build\lib\inso\Win32
**Warning:** This is not the standard location as installed by IBM Content Navigator (ECMClient\viewingservices\nativ
that HTML or PDF conversion might not function

## User Agent

Your user agent header is: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chro
Safari/536.5

## Verify Client Side Plugins

| Plugin | Status |
|---|---|
| IBM AFP Viewer Plugin | installed |
| Adobe Reader | installed |

*Figure 14-4 Viewer verification*

## 14.6 Conclusion

This chapter describes debugging and logging. We focus on how to identify important information and how to gather it for troubleshooting.

# Part 4

# Appendixes

This part lists the privileges that are necessary when you create an action. It describes the class and properties of the example document class that the example plug-ins in this book use. It also describes how locate and use the additional materials that are provided with this book. This part contains the following appendix sections:


- Appendix A, “Action privileges” on page 513
- Appendix B, “Document class definition” on page 515
- Appendix C, “Core code for the custom search plug-in project” on page 519
- Appendix D, “Additional material” on page 535


© Copyright IBM Corp. 2012, 2014. All rights reserved.

A

# Action privileges

When you create an action, you can specify the necessary privileges to execute the action. This task is done by returning one of the privileges from Table A-1. The action is enabled only if the user has all the specified privileges for all of the currently selected objects. If no special privileges are required to execute the action, then the getPrivileges() method returns an empty string.

The privileges in Table A-1 are repository-neutral and are mapped to the appropriate underlying permissions from the current repository. For example, if you are working on a FileNet P8 repository, and the action is associated with the privEditProperties privilege, then the action is enabled only if the current user has the appropriate FileNet P8 privileges to edit the properties for all of the currently selected objects.

*Table A-1 Acton privileges*

| Privilege | Description |
|---|---|
| privEditProperties | The user must have privileges to modify properties on the selected objects. |
| privEditDoc | The user must have privileges to edit the selected document. |
| privViewNotes | The user must have privileges to view notes attached to the object. |
| privAddDoc | The user must have privileges to add a document. |
| privAddItem | The user must have privileges to add an item. |

© Copyright IBM Corp. 2012, 2014. All rights reserved.

| Privilege | Description |
|---|---|
| privEmailDoc | The user must have privileges to email a document. |
| privExport | The user must have privileges to export the object. |
| privAddToFolder | The user must have privileges to add a document to the folder. |
| privRemoveFromFolder | The user must have privileges to removed a document from the folder. |
| privAddLink | The user must have privileges to add a link. |
| privRemoveLink | The user must have privileges to remove a link. |
| privAddNotes | The user must have privileges to add a note. |
| privPrintNotes | The user must have privileges to print a note. |
| privPrintDoc | The user must have privileges to print a document. |
| privCheckInOutDoc | The user must have privileges to check in and check out a document. |
| privCheckInDoc | The user must have privileges to check in a document |
| privCheckOutDoc | The user must have privileges to check out the selected documents. |
| privCancelCheckOutDoc | The user must have privileges to cancel check-out of the selected documents. |
| privViewAnnotations | The user must have privileges to view annotations on the selected documents. |
| privEditAnnotations | The user must have privileges to edit annotations on the selected documents. |
| privDelete | The user must have privileges to delete the selected objects. |
| privStartWorkflow | The user must have privileges to start a workflows. |
| privHold | The user must have privileges to place a hold. |
| privMoveToFolder | The user must have privileges to move the selected folders. |
| privChangeClass | The user must have privileges to change the class of the selected documents. |
| privMajorVersion | The user must have privileges to create a major version of the selected documents. |
| privMinorVersion | The user must have privileges to create a minor version of the selected documents. |

B

# Document class definition

The example plug-ins in this book use an example document class. This appendix describes the class and properties. See Appendix D, “Additional material” on page 535 for details of how to copy a definition that can be used with either a FileNet Content Manager or IBM Content Manager system.

The purpose of the class is to enable the example plug-ins to function correctly; it is not intended to be used as a sample in a production system.

You may, however, alter the examples to work with your own document classes or item types.

For the downloadable files that are described here, see Appendix D, “Additional material” on page 535.

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# Adding example class to IBM FileNet Content Manager

You can either create your own document class by using IBM FileNet Enterprise Manager or use the export manifest that is provided in the downloadable files.

To import the document classes, complete the following steps:

1. Download the example IBM FileNet P8 document class export and extract it to the `C:` drive.
2. Launch FileNet Enterprise Manager.
3. Select the target object store where you want to create the document class.
4. Right-click and select **All Tasks** → **Import All**.
5. Enter the following text:

   ```
   C:\Temp\P8NavigatorExample_DocumetClass\NavigatorExample_CEExport_Ma
   nifest.xml as the Import Manifest File
   ```

6. Select the *import* standard options, as shown in Figure B-1.
7. Click **Import**.

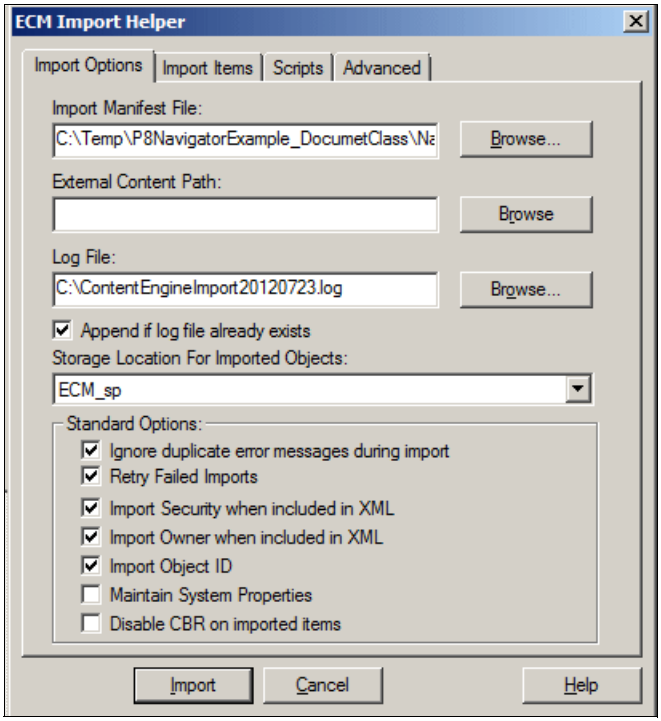



*Figure B-1 Import P8 document class*

# Adding example class to IBM Content Manager

You can either create your own document class by using the IBM Content Manager System Administration Client or use the XSD file provided in the downloadable files.

To add the example item type, complete the following steps:

1. Download the example IBM Content Manager item type XSD file to the `C:` drive.
2. Launch the IBM Content Manager System Administration Client.
3. Select **Tools** → **Import XML**.
4. Enter `C:\InsuranceDocuments.XSD` as the Data Model File, as shown in Figure B-2.
5. Select **Process interactively**.
6. Click **Import**.

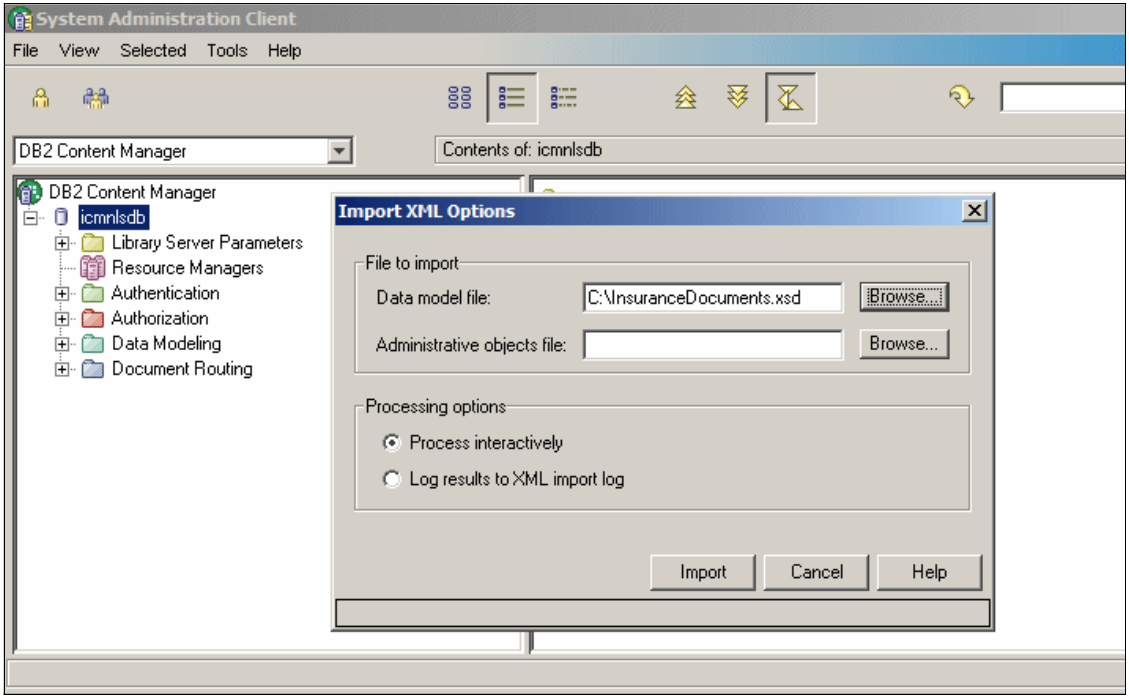



*Figure B-2 Import example Content Manager Item type*

C

# Core code for the custom search plug-in project

You can get all the code from the custom search plug-in project with this book as the additional material to be downloaded from the website.

For your convenience, here are code for two core files:

- VirtualFolderBrowsePane.js
- Enhanced SamplePluginSearchServiceP8.java

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# VirtualFolderBrowsePane.js

The complete code for VirtualFolderBrowserPane.js is shown in Example C-1.

*Example C-1 VirtualFolderBrowsePane.js*

```
define([
    "dojo/_base/declare",
    "dojo/_base/lang",
    "dojo/dom-construct",
    "idx/layout/BorderContainer",
    "dijit/layout/ContentPane",
    "dojo/json",
    "dojo/store/Memory",
     "dijit/tree/ObjectStoreModel",
    "dijit/Tree",
    "ecm/model/Request",
    "ecm/model/ResultSet",
    "ecm/widget/layout/_LaunchBarPane",
    "ecm/widget/layout/_RepositorySelectorMixin",
    "ecm/widget/listView/ContentList",
    "ecm/widget/listView/gridModules/RowContextMenu",
    "ecm/widget/listView/modules/Toolbar",
    "ecm/widget/listView/modules/Breadcrumb",
    "ecm/widget/listView/modules/InlineMessage",
    "ecm/widget/listView/modules/TotalCount",
    "ecm/widget/listView/modules/FilterData",
    "ecm/widget/listView/modules/DocInfo",
    "ecm/widget/listView/gridModules/DndRowMoveCopy",
    "ecm/widget/listView/gridModules/DndFromDesktopAddDoc",
    "ecm/widget/listView/modules/Bar",
    "ecm/widget/listView/modules/ViewDetail",
    "ecm/widget/listView/modules/ViewMagazine",
    "ecm/widget/listView/modules/ViewFilmStrip",
    "dojo/text!./templates/VirtualFolderBrowsePane.html"
],

function(declare,
       lang,
       domConstruct,
       idxBorderContainer,
       ContentPane,
       json,
       Memory,
       ObjectStoreModel,
       Tree,
       Request,
       ResultSet,
       _LaunchBarPane,
       _RepositorySelectorMixin,
```

```
        ContentList,
        RowContextMenu,
        Toolbar,
        Breadcrumb,
        InlineMessage,
        TotalCount,
        FilterData,
        DocInfo,
        DndRowMoveCopy,
        DndFromDesktopAddDoc,
        Bar,
        ViewDetail,
        ViewMagazine,
        ViewFilmStrip,
        template) {

    /**
     * @name customSearchPluginDojo.VirtualFolderBrowsePane
     * @class
     * @augments ecm.widget.layout._LaunchBarPane
     */
    return declare("customSearchPluginDojo.VirtualFolderBrowsePane", [
        _LaunchBarPane,
        _RepositorySelectorMixin
    ], {
        /** @lends customSearchPluginDojo.VirtualFolderBrowsePane.prototype */

        templateString: template,
        widgetsInTemplate: true,
        classnames:[],
        mainfolders:[],
        createdTreeItems:false,

        postCreate: function() {
            this.logEntry("postCreate");
            this.inherited(arguments);
            this.defaultLayoutRepositoryComponent = "others";
            this.setRepositoryTypes("cm,p8");
            this.createRepositorySelector();
            this.doRepositorySelectorConnections();

            // If there is more than one repository in the list, show the selector to
the user.
            if (this.repositorySelector.getNumRepositories() > 1) {
                domConstruct.place(this.repositorySelector.domNode,
this.repositorySelectorArea, "only");
            }
            this.logExit("postCreate");
        },

        /**
```

```
 * Returns the content list grid modules used by this view.
 *
 * @return Array of grid modules.
 */
getContentListGridModules: function() {
   var array = [];
   array.push(DndRowMoveCopy);
   array.push(DndFromDesktopAddDoc);
   array.push(RowContextMenu);
   return array;
},

/**
 * Returns the content list modules used by this view.
 *
 * @return Array of content list modules.
 */
getContentListModules: function() {
   var viewModules = [];
   viewModules.push(ViewDetail);
   viewModules.push(ViewMagazine);
   if (ecm.model.desktop.showViewFilmstrip) {
      viewModules.push(ViewFilmStrip);
   }

   var array = [];
   array.push(DocInfo);
   array.push({
      moduleClass: Bar,
      top: [
             [
                [
                   {
                      moduleClass: Toolbar
                   },
                   {
                      moduleClass: FilterData
                   },
                   {
                      moduleClasses: viewModules,
                      "className": "BarViewModules"
                   }
                ]
             ],
             [
                [
                   {
                      moduleClass: Breadcrumb,
                      rootPrefix:  "Virtual folder"
                   }
                ]
```

```
                    ],
                    [
                        [
                            {
                                moduleClass: InlineMessage,
                                "className": "inlineMessage"
                            }
                        ]
                    ]
                ],
                bottom: [
                    [
                        [
                            {
                                moduleClass: TotalCount
                            }
                        ]
                    ]
                ]
            });
            return array;
        },

        /**
         * Loads the content of the pane. This is a required method to insert a pane
into the LaunchBarContainer.
         */
        loadContent: function() {
            this.logEntry("loadContent");
            if (!this.repository) {
                this.setPaneDefaultLayoutRepository();
            }
            var data ="{\"name\": \"Multiple Demension Tree\",\"id\":
\"root\",\"children\": [{\"name\": \"My Navigator\",\"id\": \"my_navigator\",
\"children\":[]}]}";
            this.TreeStore = new Memory({
                 data: [ json.parse(data) ],
                 getChildren: lang.hitch(this,function(object){
                        return object.children;
                 })
             });
            this._resetTree();
            this.navTree.placeAt(dijit.byId("navTreePane").containerNode);
          var callbackGetFolders = lang.hitch(this, function(resultset)
          {
           this.mainfolders=[];
            for(row in resultset.items)
            {
                var element= {
                       value:resultset.items[row].id,
                       label:resultset.items[row].name,
```

```
                    name:"Folder: "+resultset.items[row].name,
                    id:resultset.items[row].id,
                    criterionType:"Folder",
                    children:[]
                }
                this.mainfolders.push(element);
            }
            this.setupNavTree();
        });
        var callbackGetClasses = lang.hitch(this, function(contentClasses)
         {
                this.classnames=[];
            for(docclass in contentClasses)
            {
                var element= {
                    value:contentClasses[docclass].id,
                    label:contentClasses[docclass].name,
                    name:"Class: "+contentClasses[docclass].name,
                    id:contentClasses[docclass].id,
                    criterionType:"Class",
                    children:[]
                }
                this.classnames.push(element);
            }
         });

            if (this.repository && this.repository.canListFolders()) {
                var rootItemId = this.repository.rootFolderId || "/";
                var _this = this;
                this.repository.retrieveContentClasses(callbackGetClasses);
                this.repository.retrieveItem(rootItemId, lang.hitch(this,
function(rootFolder) {
                    this.repository.rootFolder=rootFolder;
                    rootFolder.retrieveFolderContents(true, callbackGetFolders);
                }), null, null, null, this._objectStore ? this._objectStore.id : "");
            }
            this.isLoaded = true;
            this.needReset = false;
            this.logExit("loadContent");
        },
        _resetTree:function()
        {
            if(this.navTree)
                this.navTree.destroy();
            TreeModel = new ObjectStoreModel({
                 store: this.TreeStore,
                 query: {id: 'root'},
                 mayHaveChildren: function(item){
                     return "children" in item;
                 }
             });
```

```
            this.navTree = new Tree({
                model: TreeModel,
                onOpenClick: true,
                persist: false,
                getIconClass: lang.hitch(this, function(item,opened)
                    {
                        if(item.id != "root" && item.id !="my_navigator")
                            return (opened ? "searchFolderOpenIcon" :
"searchFolderCloseIcon");
                        else
                            return (opened ? "dijitFolderOpened" :
"dijitFolderClosed");
                    })
            }, "divTree");
            this.navTree.domNode.style.height="100%";
            this.navTree.placeAt(dijit.byId("navTreePane").containerNode);
            this.createdTreeItems=false;
            this.connect(this.navTree, "onClick", lang.hitch(this, function(item) {
                 if(item.id != "root" && item.id !="my_navigator")
                {
                    this.executeSearch(item);
                }
            }));
        },
        setupNavTree:function()
        {
            if(!this.createdTreeItems)
            {
                var firstLayer=this.mainfolders;
                var secondlayer=this.classnames;
                if(secondlayer.length>0)
                {
                    for(j in firstLayer)
                    {
                        firstLayer[j].children=secondlayer;
                    }
                }
                this.TreeStore.data[0].children[0].children=firstLayer;//it's the path
of "my navigator";
                this.createdTreeItems=true;
            }
        },
        executeSearch:function(item){
            var node = this.navTree.getNodesByItem(item);
            var path = node[0].tree.path;
            var docClassName="";
            var mainFolderID="";
            for(i=2;path[i]!=undefined;i++)
            {
                if(path[i].criterionType=="Class")
                {
```

```
                docClassName=path[i].value;
            }else if(path[i].criterionType == "Folder")
            {
                mainFolderID=path[i].value;
            }
         }
         this.runSearch(docClassName,mainFolderID);
      },
      /**
       * Sets the repository being used for search.
       *
       * @param repository
       *        An instance of {@link ecm.model.Repository}
       */
      setRepository: function(repository) {
         this.repository = repository;
         if (this.repositorySelector && this.repository) {
            this.repositorySelector.getDropdown().set("value",
this.repository.id);
         }
         this.navResult.reset();
         this.loadContent();
      },
      runSearch:function(docClassName,mainFolderID,attributeName,attributeValue){
         var requestParams = {};
         requestParams.repositoryId = this.repository.id;
         requestParams.repositoryType = this.repository.type;
         if( this.repository.type == "cm" ){
            var scoperule="/* ";
            var baserulestart='[(@SEMANTICTYPE IN (1))';
            var ruleEnd="]"
            var folderrule='INBOUNDLINK[@LINKTYPE = "DKFolder"]/@SOURCEITEMREF =
';
            var attributerule="";
            if(docClassName!="")
               scoperule='/'+docClassName+" ";
            var query = scoperule+baserulestart;
            if(attributeName!="" && attributeName!=undefined)
            {
               attributerule='((@' +attributeName+" = "+attributeValue +'))'
               query = query +" AND "+attributerule;
            }
            if(mainFolderID!="")
            {
                  itemid =mainFolderID.split(" ")[6];
                  mainFolderID =itemid.substr(0, itemid.length-2);
               folderrule = folderrule+'"'+mainFolderID+'"';
               query = query +" AND "+folderrule;
            }
            query +=ruleEnd;
            requestParams.query = query;
```

```
          }else if(this.repository.type=="p8"){
              var query = "Select * from ";
              if ( docClassName && docClassName.length > 0 ){
                  query += docClassName;
              }else{
                  query += "DOCUMENT ";
              }
              if( mainFolderID || ( attributeName && attributeValue) ){
                  query += " where "
              }
              if( mainFolderID && mainFolderID.length >0 ){
                  var folderID = mainFolderID.substr(  mainFolderID.length-38,
mainFolderID.length );
                  query += " this INFOLDER " + folderID ;
              }
              requestParams.query=query;
          }


          Request.invokePluginService("CustomSearchPlugin", "SearchService",
              {
                  requestParams: requestParams,
                  requestCompleteCallback: lang.hitch(this, function(response) {//
success
                      response.repository = this.repository;
                      var resultSet = new ResultSet(response);

this.navResult.setContentListModules(this.getContentListModules());

this.navResult.setGridExtensionModules(this.getContentListGridModules());
                      this.navResult.setResultSet(resultSet);

                      var inlineMessageModule = this.navResult.getContentListModule(
"inlineMessage" );
                      if (inlineMessageModule){
                          inlineMessageModule.clearMessage();
                          inlineMessageModule.setMessage("Result set items length is:
" +resultSet.items.length, "info");
                      }
                  })
              }
          );
       }
   });
});
```

# Enhanced SamplePluginSearchServiceP8.java code

The complete code for the enhanced SamplePluginSearchServiceP8.java is shown in Example C-2.

*Example C-2 SamplePluginSearchServiceP8.java*

```
package com.ibm.ecm.extension.customsearch;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Locale;

import javax.servlet.http.HttpServletRequest;

import org.apache.struts.util.MessageResources;

import com.filenet.api.collection.IndependentObjectSet;
import com.filenet.api.collection.PageIterator;
import com.filenet.api.core.Document;
import com.filenet.api.core.ObjectStore;
import com.filenet.api.property.PropertyFilter;
import com.filenet.api.query.SearchSQL;
import com.filenet.api.query.SearchScope;

import com.ibm.ecm.configuration.Config;
import com.ibm.ecm.configuration.RepositoryConfig;
import com.ibm.ecm.extension.PluginServiceCallbacks;
import com.ibm.ecm.json.JSONResultSetColumn;
import com.ibm.ecm.json.JSONResultSetResponse;
import com.ibm.ecm.json.JSONResultSetRow;
import com.ibm.json.java.JSONArray;
import com.ibm.json.java.JSONObject;

/**
 * This class contains P8 specific logic for the sample plugin search service. It
demonstrates running a search using
 * the P8 APIs and populating a JSONResultSetResponse object, which is used to
populate the ecm.model.ResultSet
 * JavaScript model class. This class provides the structure and rows for the
ecm.widget.listView.ContentList DOJO
 * widget.
 */
public class SamplePluginSearchServiceP8 {
    public static final int pageSize = 50;
    public static int totalCount = 0;

    /**
     * Runs the P8 search SQL
```

```
    *
    * @param objectStore
    *            Handle to the ObjectStore
    * @param query
    *            The query to run
    * @param callbacks
    *            The PluginServiceCallbacks object
    * @param jsonResultSet
    *            JSONResultSetResponse to build up the grid structure and rows.
    * @param clientLocale
    *            The locale of the client
    */
   public static void executeP8Search(HttpServletRequest request, String
repositoryId, String query, PluginServiceCallbacks callbacks, JSONResultSetResponse
jsonResultSet, Locale clientLocale) throws Exception {
       ObjectStore objectStore = callbacks.getP8ObjectStore(repositoryId);

       buildP8ResultStructure(request, jsonResultSet, callbacks.getResources(),
clientLocale);

       //assume the query string has no OPTIONS.
       if( ifCE52OrAbove(objectStore) ){
          query += " OPTIONS (COUNT_LIMIT " + Integer.MAX_VALUE + ") ";
       }
       Logger.logEntry(SamplePluginSearchServiceP8.class, "executeP8Search",
request, "Query String: " + query);

       SearchSQL searchSQL = new SearchSQL(query);
       SearchScope searchScope = new SearchScope(objectStore);

       // Use callbacks.getP8FolderResultsPropertyFilter() for folder results.
       PropertyFilter filter = callbacks.getP8DocumentResultsPropertyFilter();

       // Retrieve the first pageSize results.
       List<Object> searchResults = new ArrayList<Object>(pageSize);
       IndependentObjectSet resultsObjectSet = searchScope.fetchObjects(searchSQL,
pageSize, filter, true);
       PageIterator pageIterator = resultsObjectSet.pageIterator();

       int itemCount = 0;

       if (pageIterator.nextPage()) {
          for (Object obj : pageIterator.getCurrentPage()) {
             searchResults.add(obj);
             itemCount++;
          }

       }
       if (itemCount < pageSize){
          totalCount = itemCount;
       }else{
```

```
            //this just works for CE 5.2 or above
            if (pageIterator != null && ifCE52OrAbove( objectStore )) {
                Integer totalCountInteger = pageIterator.getTotalCount();
                System.out.println( totalCountInteger );
                if( totalCountInteger!=null ){
                    totalCount = totalCountInteger.intValue();
                    if( totalCount < 0 ) totalCount = -totalCount;//means reach CE
threshold
                }
            }
        }
        // Retrieve the privilege masks for the search results.
        HashMap<Object, Long> privMasks =
callbacks.getP8PrivilegeMasks(repositoryId, searchResults);

        for (Object searchResult : searchResults) {
            Document doc = (Document) searchResult;
            /*
             *  IDs use the form:
             *  <object class name>,<object store ID>,<object ID>
             */
            StringBuffer sbId = new StringBuffer();

sbId.append(doc.getClassName()).append(",").append(objectStore.get_Id().toString())
.append(",").append(doc.get_Id().toString());

            long privileges = (privMasks != null) ? privMasks.get(doc) : 0L;

            JSONResultSetRow row = new JSONResultSetRow(sbId.toString(),
doc.get_Name(), doc.get_MimeType(), privileges);

            // Add locked user information (if any)
            row.addAttribute("locked", doc.isLocked(), JSONResultSetRow.TYPE_BOOLEAN,
null, (new Boolean(doc.isLocked())).toString());
            row.addAttribute("lockedUser", doc.get_LockOwner(),
JSONResultSetRow.TYPE_STRING, null, doc.get_LockOwner());
            row.addAttribute("currentVersion", doc.get_IsCurrentVersion(),
JSONResultSetRow.TYPE_BOOLEAN, null, (new
Boolean(doc.get_IsCurrentVersion())).toString());

            // Add the attributes
            row.addAttribute("ID", doc.get_Id().toString(),
JSONResultSetRow.TYPE_STRING, null, doc.get_Id().toString());
            row.addAttribute("className", doc.getClassName(),
JSONResultSetRow.TYPE_STRING, null, doc.getClassName());
            row.addAttribute("ModifiedBy", doc.get_LastModifier(),
JSONResultSetRow.TYPE_STRING, null, doc.get_LastModifier());
            row.addAttribute("LastModified", doc.get_DateLastModified().toString(),
JSONResultSetRow.TYPE_TIMESTAMP, null, doc.get_DateLastModified().toString());
```

```
            row.addAttribute("Version", doc.get_MajorVersionNumber() + "." +
doc.get_MinorVersionNumber(), JSONResultSetRow.TYPE_STRING, null,
doc.get_MajorVersionNumber() + "." + doc.get_MinorVersionNumber());
            row.addAttribute("{NAME}", doc.get_Name(), JSONResultSetRow.TYPE_STRING,
null, doc.get_Name());
            row.addAttribute("ContentSize", doc.get_ContentSize(),
JSONResultSetRow.TYPE_INTEGER, null, null);

            jsonResultSet.addRow(row);
        }
        String sessionKey = "pagerIterator";
        request.getSession().removeAttribute(sessionKey);
        if (itemCount == pageSize) {
            jsonResultSet.put("continuationData", sessionKey);
            //this require CE version >=5.0
            request.getSession().setAttribute(sessionKey, pageIterator);
        }
        if( totalCount > 0 ){
            jsonResultSet.put("totalCount", totalCount);
            jsonResultSet.put("totalCountType", "total");
        }

    }

    /**
     * Builds the details and magazine structure for P8. This method will use a set
of predefined columns and fields
     * that always exist on every P8 object.
     *
     * @param jsonResultSet
     *            The JSONResultSetResponse object to populate with the structure
     * @param messageResources
     *            The resource bundle to retrieve default column names
     * @param clientLocale
     *            The locale of the client
     */
    private static void buildP8ResultStructure(HttpServletRequest request,
JSONResultSetResponse jsonResultSet, MessageResources resources, Locale
clientLocale) {
        RepositoryConfig repoConf = Config.getRepositoryConfig(request);
        String[] folderColumns = repoConf.getSearchDefaultColumns();

        String[] states = new String[1];
        states[0] = JSONResultSetColumn.STATE_LOCKED;

        jsonResultSet.addColumn(new JSONResultSetColumn("&amp;nbsp;", "multiStateIcon",
false, states));
        jsonResultSet.addColumn(new JSONResultSetColumn("&amp;nbsp;", "17px",
"mimeTypeIcon", null, false));
```

```
       //jsonResultSet.addColumn(new
JSONResultSetColumn(resources.getMessage(clientLocale, "search.results.header.id"),
"200px", "ID", null, false));
       //jsonResultSet.addColumn(new JSONResultSetColumn("Class Name", "125px",
"className", null, false));
       for (String columnString : folderColumns) {
          if (columnString.equals("LastModifier"))
             jsonResultSet.addColumn(new
JSONResultSetColumn(resources.getMessage(clientLocale,
"search.results.header.lastModifiedByUser"), "125px", "ModifiedBy", null, false));
          else if (columnString.equals("DateLastModified"))
             jsonResultSet.addColumn(new
JSONResultSetColumn(resources.getMessage(clientLocale,
"search.results.header.lastModifiedTimestamp"), "175px", "LastModified", null,
false));
          else if (columnString.equals("MajorVersionNumber"))
             jsonResultSet.addColumn(new
JSONResultSetColumn(resources.getMessage(clientLocale,
"search.results.header.version"), "50px", "Version", null, false));
          else if (columnString.equals("{NAME}"))
             jsonResultSet.addColumn(new JSONResultSetColumn("Name", "200px",
columnString, null, false));
          else {
             jsonResultSet.addColumn(new JSONResultSetColumn(columnString, "80px",
columnString, null, true));
          }
       }
       // Magazine view
       jsonResultSet.addMagazineColumn(new JSONResultSetColumn("thumbnail", "60px",
"thumbnail", null, null));

       JSONArray fieldsToDisplay = new JSONArray();
       JSONObject jsonObj = new JSONObject();
       jsonObj.put("field", "className");
       jsonObj.put("displayName", "Class");
       fieldsToDisplay.add(jsonObj);

       jsonObj = new JSONObject();
       jsonObj.put("field", "ModifiedBy");
       jsonObj.put("displayName", resources.getMessage(clientLocale,
"search.results.header.lastModifiedByUser"));
       fieldsToDisplay.add(jsonObj);

       jsonObj = new JSONObject();
       jsonObj.put("field", "LastModified");
       jsonObj.put("displayName", resources.getMessage(clientLocale,
"search.results.header.lastModifiedTimestamp"));
       fieldsToDisplay.add(jsonObj);

       jsonObj = new JSONObject();
       jsonObj.put("field", "Version");
```

```
      jsonObj.put("displayName", resources.getMessage(clientLocale,
"search.results.header.lastModifiedTimestamp"));
      fieldsToDisplay.add(jsonObj);

      jsonResultSet.addMagazineColumn(new JSONResultSetColumn("content", "100%",
"content", fieldsToDisplay, null));
   }


   public static boolean ifCE52OrAbove(ObjectStore os) {
      boolean result = false;
      if (os != null) {
         String schema = os.getProperties().getStringValue("SchemaVersion");
         int index = schema.indexOf(".");
         if (index < 0) {
            index = schema.length();
         }
         String mainNumber = schema.substring(0, index);
         if (Integer.parseInt(mainNumber) >= 18) {
            result = true;
         }
      }
      return result;
   }
```

# D

# Additional material

This book refers to additional material that can be downloaded from the Internet as described in the following sections.

## Locating the web material

The web material associated with this book is available in softcopy on the Internet from the IBM Redbooks web server. Point your web browser at:

`ftp://www.redbooks.ibm.com/redbooks/SG248055`

Alternatively, you can go to the IBM Redbooks website at:

`ibm.com/redbooks`

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG24-8055.

© Copyright IBM Corp. 2012, 2014. All rights reserved.

# Using the web material

The additional web material that accompanies this book includes the following files:

| *File name* | *Description* |
| --- | --- |
| **SG248055.zip** | Compressed code samples |
| SG24-8055-00.pdf | Previous version of this IBM Redbooks publication |

## System requirements for downloading the web material

The web material requires the following system configuration:

**Hard disk space**: 20 MB minimum

## Downloading and extracting the web material

Create a subdirectory (folder) on your workstation, and extract the contents of the web material from the `.zip` file into this folder.

# Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

## IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- *Advanced Case Management with IBM Case Manager,* SG24-7929
- *IBM Content Analytics Version 2.2: Discovering Actionable Insight from Your Content*, SG24-7877
- *IBM FileNet Content Manager Implementation Best Practices and Recommendations*, SG24-7547
- *IBM FileNet P8 Platform and Architecture*, SG24-7667
- *Understanding IBM FileNet Records Manager*, SG24-7623 (IBM FileNet Records Manager is currently known as IBM Enterprise Records)

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

**ibm.com**/redbooks

## Online resources

These websites are also relevant as further information sources:

- IBM Content Manager with IBM Content Navigator:

  http://publib.boulder.ibm.com/infocenter/cmgmt/v8r5m0
- IBM Content Manager OnDemand with IBM Content Navigator:

  http://pic.dhe.ibm.com/infocenter/cmod/v9r0m0
- IBM FileNet Content Manager with IBM Content Navigator:

  http://publib.boulder.ibm.com/infocenter/p8docs/v5r2m0

© Copyright IBM Corp. 2012, 2014. All rights reserved.

- Hardware and software requirements for IBM Content Navigator:

  http://www.ibm.com/support/docview.wss?uid=swg27024958
- IBM Content Navigator public forum:

  http://www.ibm.com/developerworks/forums/forum.jspa?forumID=2869&cat=19
- IBM Content Navigator publication library:

  http://www.ibm.com/support/docview.wss?uid=swg27025015
- A complete Java API reference:

  http://pic.dhe.ibm.com/infocenter/p8docs/v5r2m0/topic/com.ibm.javaeuc.doc/overview-summary.html
- A complete JavaScript API reference:

  http://pic.dhe.ibm.com/infocenter/p8docs/v5r2m0/topic/com.ibm.developingeuc.doc/doc/JavaScriptdoc/index.html
- Rational Application Developer Version 9:

  http://publib.boulder.ibm.com/infocenter/radhelp/v9/index.jsp
- WebSphere Application Server Version 8:

  http://pic.dhe.ibm.com/infocenter/wasinfo/v8r0
- WebSphere Application Server Version 8.5:

  http://pic.dhe.ibm.com/infocenter/wasinfo/v8r5
- IBM Worklight Version 6.0.0:

  http://pic.dhe.ibm.com/infocenter/wrklight/v6r0m0

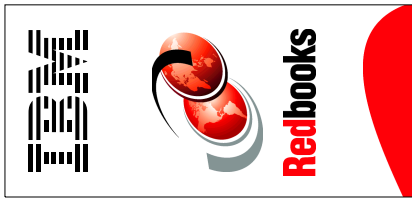
# Help from IBM

IBM Support and downloads

**ibm.com**/support

IBM Global Services

**ibm.com**/services



# Customizing and Extending IBM Content Navigator

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

# Customizing and Extending IBM Content Navigator

**Understand extension points and customization options**

**Create an action, service, feature, and custom step processor**

**Use widgets in apps, mobile development, and more**

IBM Content Navigator provides a unified user interface for your Enterprise Content Management (ECM) solutions. It also provides a robust development platform so you can build customized user interface and applications to deliver value and an intelligent, business-centric experience.

This IBM Redbooks publication guides you through the Content Navigator platform, its architecture, and the available programming interfaces. It describes how you can configure and customize the user interface with the administration tools provided, and how you can customize and extend Content Navigator using available development options with sample code. Specifically, the book shows how to set up a development environment, and develop plug-ins that add an action, service, and feature to the user interface. Customization topics include implementing request and response filters, external data services (EDS), creating custom step processors, and using Content Navigator widgets in other applications. This book also covers mobile development, viewer customization, component deployment, and debugging and troubleshooting.

This book is intended for IT architects, application designers and developers working with IBM Content Navigator and IBM ECM products. It offers a high-level description of how to extend and customize IBM Content Navigator and also more technical details of how to do implementations with sample code.

**INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION**

**BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:**
**ibm.com**/redbooks

SG24-8055-01 ISBN 0738439215